Arm® A64 Instruction Set Architecture
Armv8, for Armv8-A architecture profile
Arm A64 Instruction Set Architecture
Armv8, for Armv8-A architecture profile

Copyright © 2010-2020 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 ISA XML for Armv8.7 (2020-09).

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.

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 affiliates) 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 © 2010-2020 Arm Limited (or its affiliates). All rights reserved.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349 version 21.0)

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

The information in this document relating to v8.7 of the architecture and the features introduced in this release is at Alpha quality. Alpha quality means that most major features of the specification are included, features and details might be missing.

The information in this document relating to versions of the architecture before v8.7 and features introduced in previous releases is at Beta quality. Beta quality means that all major features of the specification are included, some details might be missing.
Web Address

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

ADC: Add with Carry.
ADCS: Add with Carry, setting flags.
ADD (extended register): Add (extended register).
ADD (immediate): Add (immediate).
ADD (shifted register): Add (shifted register).
ADDG: Add with Tag.
ADDS (extended register): Add (extended register), setting flags.
ADDS (immediate): Add (immediate), setting flags.
ADDS (shifted register): Add (shifted register), setting flags.
ADR: Form PC-relative address.
ADR: Form PC-relative address to 4KB page.
AND (immediate): Bitwise AND (immediate).
ANDS (immediate): Bitwise AND (immediate), setting flags.
ANDS (shifted register): Bitwise AND (shifted register), setting flags.
ASR (immediate): Arithmetic Shift Right (immediate): an alias of SBFM.
ASR (register): Arithmetic Shift Right (register): an alias of ASRV.
ASRV: Arithmetic Shift Right Variable.
AT: Address Translate: an alias of SYS.
AUTDA, AUTDZA: Authenticate Data address, using key A.
AUTDB, AUTDZB: Authenticate Data address, using key B.
AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA: Authenticate Instruction address, using key A.
AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB: Authenticate Instruction address, using key B.
AXFLAG: Convert floating-point condition flags from Arm to external format.
B: Branch.
B.cond: Branch conditionally.
BFC: Bitfield Clear: an alias of BFM.
BFI: Bitfield Insert: an alias of BFM.
BFM: Bitfield Move.
BFXIL: Bitfield extract and insert at low end: an alias of BFM.
BIC (shifted register): Bitwise Bit Clear (shifted register).
BICS (shifted register): Bitwise Bit Clear (shifted register), setting flags.
BL: Branch with Link.
BLR: Branch with Link to Register.
BLRAA, BLRAAZ, BLRAB, BLRABZ: Branch with Link to Register, with pointer authentication.
BR: Branch to Register.
BRAA, BRAAZ, BRAB, BRABZ: Branch to Register, with pointer authentication.
BRK: Breakpoint instruction.
BTI: Branch Target Identification.
CAS, CASA, CASAL, CASL: Compare and Swap word or doubleword in memory.
CASB, CASAB, CASALB, CASLB: Compare and Swap byte in memory.
CASH, CASAH, CASALH, CASLH: Compare and Swap halfword in memory.
CASP, CASPA, CASPAL, CASPL: Compare and Swap Pair of words or doublewords in memory.
CBNZ: Compare and Branch on Nonzero.
CBZ: Compare and Branch on Zero.
CCMN (immediate): Conditional Compare Negative (immediate).
CCMN (register): Conditional Compare Negative (register).
CCMP (immediate): Conditional Compare (immediate).
CCMP (register): Conditional Compare (register).
CFINV: Invert Carry Flag.
CFP: Control Flow Prediction Restriction by Context: an alias of SYS.
CINC: Conditional Increment: an alias of CSINC.
CINV: Conditional Invert: an alias of CSINV.
CLREX: Clear Exclusive.
CLS: Count Leading Sign bits.
CLZ: Count Leading Zeros.
CMN (immediate): Compare Negative (immediate): an alias of ADDS (immediate).
CMP (immediate): Compare (immediate): an alias of SUBS (immediate).
CMPP: Compare with Tag: an alias of SUBPS.
CNEG: Conditional Negate: an alias of CSNEG.
CPP: Cache Prefetch Prediction Restriction by Context: an alias of SYS.
CRC32CB, CRC32CH, CRC32CW, CRC32CX: CRC32C checksum.
CSDB: Consumption of Speculative Data Barrier.
CSEL: Conditional Select.
CSET: Conditional Set: an alias of CSINC.
CSETM: Conditional Set Mask: an alias of CSINV.
CSINC: Conditional Select Increment.
CSINV: Conditional Select Invert.
CSNEG: Conditional Select Negation.
DC: Data Cache operation: an alias of SYS.
DCPS1: Debug Change PE State to EL1.
DCPS2: Debug Change PE State to EL2.
DCPS3: Debug Change PE State to EL3.
DGH: Data Gathering Hint.
DMB: Data Memory Barrier.
DRPS: Debug restore process state.
DSB: Data Synchronization Barrier.
DVP: Data Value Prediction Restriction by Context: an alias of SYS.
EON (shifted register): Bitwise Exclusive OR NOT (shifted register).
EOR (immediate): Bitwise Exclusive OR (immediate).
EOR (shifted register): Bitwise Exclusive OR (shifted register).
ERET: Exception Return.
ERETAA, ERETAB: Exception Return, with pointer authentication.
ESB: Error Synchronization Barrier.
EXTR: Extract register.
GMI: Tag Mask Insert.
HINT: Hint instruction.
HLT: Halt instruction.
HVC: Hypervisor Call.
IC: Instruction Cache operation: an alias of SYS.
IRG: Insert Random Tag.
ISB: Instruction Synchronization Barrier.
LD64B: Single-copy Atomic 64-byte Load.
LDADD, LDADDA, LDADDL, 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.

**LDMSAX, LDMSAXA, LDMSAXAL, LDMSAXL**: Atomic signed maximum on word or doubleword in memory.

**LDMSAXB, LDMSAXAB, LDMSAXALB, LDMSAXLB**: Atomic signed maximum on byte in memory.

**LDMSAXH, LDMSAXAH, LDMSAXALH, LDMSAXLH**: Atomic signed maximum on halfword in memory.

**LDMSMIN, LDMSMINA, LDMSMINAL, LDMSMINI**: Atomic signed minimum on word or doubleword in memory.

**LDMSMINB, LDMSMINAB, LDMSMINALB, LDMSMINLB**: Atomic signed minimum on byte in memory.

**LDMSMINH, LDMSMINAH, LDMSMINALH, LDMSMINLH**: 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, LDUMAXI**: 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, LDUMINI**: Atomic unsigned minimum on word or doubleword in memory.

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

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

**LDUR**: Load Register (unscaled).

**LDURB**: Load Register Byte (unscaled).

**LDURH**: Load Register Halfword (unscaled).

**LDURSB**: Load Register Signed Byte (unscaled).

**LDURSH**: Load Register Signed Halfword (unscaled).

**LDURSW**: Load Register Signed Word (unscaled).

**LDXP**: Load Exclusive Pair of Registers.

**LDXR**: Load Exclusive Register.

**LDXRB**: Load Exclusive Register Byte.

**LDXRH**: Load Exclusive Register Halfword.

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

**LSLV**: Logical Shift Left Variable.

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

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

**LSRV**: Logical Shift Right Variable.

**MADD**: Multiply-Add.

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

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

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

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

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

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

**MOVK**: Move wide with keep.

**MOVN**: Move wide with NOT.

**MOVZ**: Move wide with zero.

**MRS**: Move System Register.

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

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

**MSUB**: Multiply-Subtract.

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

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

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

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

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

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

**NOP**: No Operation.

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

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

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

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

**PACDB, 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.
RETTAA, 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.
SEF8, SETF16: Evaluation of 8 or 16 bit flag values.
SEV: Send Event.
SEVL: Send Event Local.
SMADDL: Signed Multiply-Add Long.
SMC: Secure Monitor Call.
SMNEGL: Signed Multiply-Negate Long: an alias of SMSUBL.
SMSUBL: Signed Multiply-Subtract Long.
SMULH: Signed Multiply High.
SMULL: Signed Multiply Long: an alias of SMADDL.
SSBB: Speculative Store Bypass Barrier.
ST2G: Store Allocation Tags.
ST64B: Single-copy Atomic 64-byte Store without Return.
ST64BV: Single-copy Atomic 64-byte Store with Return.
ST64BV0: Single-copy Atomic 64-byte EL0 Store with Return.
STADD, STADDL: Atomic add on word or doubleword in memory, without return: an alias of LDADD, LDADDA, LDADDDL, LDADDL.

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

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

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

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

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

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

STG: Store Allocation Tag.

STGM: Store Tag Multiple.

STGP: Store Allocation Tag and Pair of registers.

STLLR: Store LORelease Register.

STLLRB: Store LORelease Register Byte.

STLLRH: Store LORelease Register Halfword.

STLR: Store-Release Register.

STLRB: Store-Release Register Byte.

STLRH: Store-Release Register Halfword.

STLUR: Store-Release Register (unscaled).

STLURB: Store-Release Register Byte (unscaled).

STLURH: Store-Release Register Halfword (unscaled).

STLXP: Store-Release Exclusive Pair of registers.

STLXR: Store-Release Exclusive Register.

STLXRB: Store-Release Exclusive Register Byte.

STLXRH: Store-Release Exclusive Register Halfword.

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

STP: Store Pair of Registers.

STR (immediate): Store Register (immediate).

STR (register): Store Register (register).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**STXR:** Store Exclusive Register.

**STXRB:** Store Exclusive Register Byte.

**STXRH:** Store Exclusive Register Halfword.

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

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

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

SUB (immediate): Subtract (immediate).

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

SUBG: Subtract with Tag.

SUBP: Subtract Pointer.

SUBPS: Subtract Pointer, setting Flags.

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

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

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

SVC: Supervisor Call.

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

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

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

SXTR: Signed Extend Byte: an alias of SBFM.

SXTH: Sign Extend Halfword: an alias of SBFM.

SXTW: Sign Extend Word: an alias of SBFM.

SYS: System instruction.

SYSL: System instruction with result.

TBNZ: Test bit and Branch if Nonzero.

TBZ: Test bit and Branch if Zero.

TLBI: TLB Invalidate operation: an alias of SYS.

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


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

UBFM: Unsigned Bitfield Move.

UBFX: Unsigned Bitfield Extract: an alias of UBFM.

UDF: Permanently Undefined.

UDIV: Unsigned Divide.

UMADDL: Unsigned Multiply-Add Long.

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

UMSUBL: Unsigned Multiply-Subtract Long.

UMULH: Unsigned Multiply High.

UMULL: Unsigned Multiply Long: an alias of UMADDL.

UXTB: Unsigned Extend Byte: an alias of UBFM.

UXTH: Unsigned Extend Halfword: an alias of UBFM.
**WFE**: Wait For Event.

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

**WFI**: Wait For Interrupt.

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

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

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

**YIELD**: YIELD.
ADC

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

<table>
<thead>
<tr>
<th>sf</th>
<th>0</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>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>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)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

```plaintext
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.

32-bit (sf == 0)

ADCS <WD>, <WN>, <WM>

64-bit (sf == 1)

ADCS <XD>, <XN>, <XM>

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

Assembler Symbols

<WD> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<WN> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<WM> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<XD> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<XN> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<XM> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

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

Operational information

If PSTATE.DIT is 1:

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

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

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

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

Assembler Symbols

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>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'.

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

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

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

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

**Operation**

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

**Operational information**

If PSTATE.DIT is 1:

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

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

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

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

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>sh</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

Assembler Symbols

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

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

Alias Conditions

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

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then 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.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| sf | 0 | 0 | 0 | 1 | 0 | 1 | 1 | shift | 0 | Rm | imm6 | Rn | Rd |

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.
**ADDG**

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

**Integer**  
(Armv8.5)

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

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

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

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

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

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

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

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

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

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>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'.

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

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

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

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

**Alias Conditions**

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

**Operation**

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

**Operational information**

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

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

Alias Conditions

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

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 0 1 0 1 0 1 1 | shift 0 | Rm | imm6 | Rn | Rd
op  S
```

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

```
    shift  <shift>
    00    LSL
    01    LSR
    10    ASR
    11    RESERVED
```

- <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           | immhi           | Rd              |

op

ADR <Xd>, <label>

integer d = UInt(Rd);
bits(64) imm;
imm = SignExtend(immhi:immlo, 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 address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

Operation

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

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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  | immhi |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

ADRP <Xd>, <label>

integer d = UInt(Rd);
bits(64) imm;
imm = SignExtend(immhi:immlo:Zeros(12), 64);

Assembler Symbols

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

Operation

bits(64) base = PC[];
base<11:0> = Zeros(12);
X[d] = base + imm;
AND (immediate)

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

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

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

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

64-bit (sf == 1)

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

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

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

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

Operational information

If PSTATE.DIT is 1:

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

\[
\begin{array}{ccccccc}
\hline
sf & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \text{shift} & 0 & Rm & \text{imm6} & Rn & Rd & N & \text{opc} & \text{N}
\end{array}
\]

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

\[
\text{AND} <Wd>, <Wn>, <Wm>, \{<shift> \#<amount>\}
\]

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

\[
\text{AND} <Xd>, <Xn>, <Xm>, \{<shift> \#<amount>\}
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf == 1 \text{ then } 64 \text{ else } 32; \\
&\text{if } sf == 0 \text{ && } \text{imm6}<5> == 1 \text{ then UNDEFINED;}
\end{align*}
\]

\[
\begin{align*}
\text{ShiftType } \text{shift_type} &= \text{DecodeShift}(\text{shift}); \\
\text{integer } \text{shift_amount} &= \text{UInt}(\text{imm6});
\end{align*}
\]

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>&lt;shift&gt;</th>
<th>\text{operand2} = \text{ShiftReg}(m, \text{shift_type}, \text{shift_amount});</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>

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

\[
\begin{align*}
\text{bits}(\text{datasize}) \text{ operand1} &= X[n]; \\
\text{bits}(\text{datasize}) \text{ operand2} &= \text{ShiftReg}(m, \text{shift_type}, \text{shift_amount}); \\
\text{result} &= \text{operand1 AND operand2}; \\
X[d] &= \text{result};
\end{align*}
\]

Operational information

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

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

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

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

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

ANDS <Xd>, <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>** 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

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

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

### Operational information

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

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

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

<table>
<thead>
<tr>
<th>sf</th>
<th>1 1</th>
<th>0 1 0 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>
</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);
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

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

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.
## ASR (register)

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

This is an alias of ASRV. This means:

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

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

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

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

is equivalent to

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

and is always the preferred disassembly.

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

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

is equivalent to

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

and is always the preferred disassembly.

### Assembler Symbols

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

### Operation

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

### Operational information

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

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

This is an alias of SBFM. This means:

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

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

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

is equivalent to

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

and is always the preferred disassembly.

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

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

is equivalent to

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

and is always the preferred disassembly.

#### Assembler Symbols

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

#### Operation

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

#### Operational information

If PSTATE.DIT is 1:

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

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

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

ASRV <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.
AT

Address Translate. For more information, see \textit{op0==0b01, cache maintenance, TLB maintenance, and address translation instructions}.

This is an alias of \texttt{SYS}. This means:

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

\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
\hline
1 1 0 1 0 1 0 0 | 0 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | op1 | 0 1 1 1 | 1 0 0 | x | op2 | Rt
\hline
L | CRn | CRm
\end{verbatim}

\texttt{AT <at_op>, <Xt>}

is equivalent to

\texttt{SYS #<op1>, C7, <Cm>, #<op2>, <Xt>}

and is the preferred disassembly when \texttt{SysOp(op1,'0111',CRm,op2) == Sys_AT}.

\textbf{Assembler Symbols}

\begin{itemize}
  \item \texttt{<at_op>} is an AT instruction name, as listed for the AT system instruction group, encoded in “op1:CRm<0>:op2”:
  \begin{verbatim}
  \begin{tabular}{llllll}
  op1 & CRm<0> & op2 & <at_op> & Architectural Feature \\
  000 & 0 & 000 & S1E1R & - \\
  000 & 0 & 001 & S1E1W & - \\
  000 & 0 & 010 & S1E0R & - \\
  000 & 0 & 011 & S1E0W & - \\
  000 & 1 & 000 & S1E1RP & FEAT_PAN2 \\
  000 & 1 & 001 & S1E1WP & FEAT_PAN2 \\
  100 & 0 & 000 & S1E2R & - \\
  100 & 0 & 001 & S1E2W & - \\
  100 & 0 & 100 & S1E1R & - \\
  100 & 0 & 101 & S1E1W & - \\
  100 & 0 & 110 & S1E0R & - \\
  100 & 0 & 111 & S1E0W & - \\
  110 & 0 & 000 & S1E3R & - \\
  110 & 0 & 001 & S1E3W & - \\
  \end{tabular}
  \end{verbatim}
  \end{itemize}

- \texttt{<op1>} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- \texttt{<Cm>} is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the "CRm" field.
- \texttt{<op2>} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- \texttt{<Xt>} is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

\textbf{Operation}

The description of \texttt{SYS} gives the operational pseudocode for this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**AUTDA, AUTDZA**

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

The modifier is:

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

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

### Integer (Armv8.3)

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

### AUTDA (Z == 0)

AUTDA <Xd>, <Xn|SP>

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

AUTDZA <Xd>

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

### Assembler Symbols

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

### Operation

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

Copyright © 2010-2020 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**

(\text{Armv8.3})

\[
\begin{array}{ccccccccccccccccccccccccccccc}
\hline
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & Z & 1 & 1 & 1 & \hline
Rn & Rd
\end{array}
\]

**AUTDB (Z == 0)**

AUTDB \(<Xd>\), \(<Xn|SP>\)

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

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);
```

Copyright © 2010-2020 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)

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

### AUTIA (Z == 0)

AUTIA <Xd>, <Xn|SP>

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

AUTIZA <Xd>

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

if !HavePACExt() then
  UNDEFINED;

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

### System
(Armv8.3)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CRm</td>
<td>op2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 © 2010-2020 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 1 0 1 0 1 1 0 0 0 0 1 0 0 Z 1 0 1 Rn Rd
```

**AUTIB (Z == 0)**

AUTIB <Xd>, <Xn|SP>

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

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 1 1 0 0 1 0 0 0 x 1 1 1 x 1 1 1 1 1
CRm op2
```
AUTIB1716 (CRm == 0001 && op2 == 110)

AUTIB1716

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

AUTIBSP

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

AUTIBZ

integer d;
integer n;
boolean source_is_sp = FALSE;

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

Assembler Symbols

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

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

Operation

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

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

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

System
(Armv8.5)

```
AXFLAG
if !HaveFlagFormatExt() then UNDEFINED;
```

Operation

```
bit Z = PSTATE.Z OR PSTATE.V;
bite C = PSTATE.C AND NOT(PSTATE.V);
PSTATE.N = '0';
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = '0';
```

| CRm     |
|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1       | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 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.

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

**Asm.** `B.<cond> <label>`

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

**Assembler Symbols**

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

**Operation**

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

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

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

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & \multicolumn{4}{c}{\text{imm26}} \\
\end{array}
\]

B \texttt{<label>}

\text{bits(64) offset} = \texttt{SignExtend(imm26:’00’, 64)};

\section*{Assembler Symbols}

\texttt{<label>} \quad \text{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.}

\section*{Operation}

\texttt{BranchTo(\texttt{PC[} + \text{offset}, \texttt{BranchType_DIR});}

\begin{footnotesize}
\begin{itemize}
\item Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
\item Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
\end{itemize}
\end{footnotesize}
BFC

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

This is an alias of BFM. This means:

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

Leaving other bits unchanged

(Armv8.2)

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

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

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

is equivalent to

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

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

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

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

is equivalent to

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

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

Assembler Symbols

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

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

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

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

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

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

Operation

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

Operational information

If PSTATE.DIT is 1:

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

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

This is an alias of BFM. This means:

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

<table>
<thead>
<tr>
<th>sf</th>
<th>0 1 1 0 1 1 0 N</th>
<th>immr</th>
<th>imms</th>
<th>!= 1111</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn</td>
</tr>
</tbody>
</table>

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

BFI \(<Wd>, <Wn>, #<\text{lsb}>, #<\text{width}>\)

is equivalent to

BFM \(<Wd>, <Wn>, #(-<\text{lsb}> \text{ MOD } 32), #(<\text{width}>-1)\)

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

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

BFI \(<Xd>, <Xn>, #<\text{lsb}>, #<\text{width}>\)

is equivalent to

BFM \(<Xd>, <Xn>, #(-<\text{lsb}> \text{ MOD } 64), #(<\text{width}>-1)\)

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

Assembler Symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

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

If \( \text{imms} \) 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 \text{BFC}, \text{BFI}, and \text{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>\text{BFC}</td>
<td>\text{Rn} == '11111' &amp;&amp; \text{UInt}(\text{imms}) &lt; \text{UInt}(\text{immr})</td>
</tr>
<tr>
<td>\text{BFI}</td>
<td>\text{Rn} != '11111' &amp;&amp; \text{UInt}(\text{imms}) &lt; \text{UInt}(\text{immr})</td>
</tr>
<tr>
<td>\text{BFXIL}</td>
<td>\text{UInt}(\text{imms}) &gt;= \text{UInt}(\text{immr})</td>
</tr>
</tbody>
</table>
Operation

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

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

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

Operational information

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

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

This is an alias of BFM. This means:

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

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

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

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

is equivalent to

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

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

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

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

is equivalent to

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

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

Assembler Symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

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

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

BIC <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;
X[d] = result;
Operational information

If PSTATE.DIT is 1:

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

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

```
<table>
<thead>
<tr>
<th>sf</th>
<th>1 1 0 1 0 1 0</th>
<th>shift</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</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 \( d = 
\)\( \text{UInt}(Rd); \)

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

integer \( m = 
\)\( \text{UInt}(Rm); \)

integer datasize = if sf == '1' then 64 else 32;

if sf == '0' && imm6<5> == '1' then UNDEFINED;

**ShiftType** shift_type = DecodeShift(shift);

integer shift_amount = \( \text{UInt}(\text{imm6}); \)

**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 = \( \text{ShiftReg}(m, \text{shift}_\text{type}, \text{shift}_\text{amount}); \)

operand2 = \( \text{NOT}(\text{operand2}); \)

result = operand1 AND operand2;

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

\( X[d] = \text{result}; \)
Operational information

If PSTATE.DIT is 1:

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

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

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

\[
\begin{array}{cccccccccccccccccccc}
\mid & 0 & 0 & 1 & 0 & 1 & \mid & \text{imm26}
\end{array}
\]

\text{op}

\text{BL} \ <\text{label}>

\begin{verbatim}
bits(64) offset = SignExtend(imm26:'00', 64);
\end{verbatim}

\textbf{Assembler Symbols}

\text{<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.

\textbf{Operation}

\begin{verbatim}
X[30] = PC[] + 4;
BranchTo(PC[] + offset, BranchType_DIRCALL);
\end{verbatim}
BLR

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

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

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;

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

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

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

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

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

Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ.

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

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

### Integer (Armv8.3)

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

#### 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;

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

Branch to Register branches unconditionally to an address in a register, with a hint that this is not 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   | 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  | 0  | 0  | 0  | 0  |

BR <Xn>

integer n = UInt(Rn);

Assembler Symbols

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

Operation

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

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

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

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

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

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

Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ. If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

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

<table>
<thead>
<tr>
<th>Integer</th>
<th>(Armv8.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Integer Table" /></td>
<td></td>
</tr>
</tbody>
</table>

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

```
BRAAZ <Xn>
```

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

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

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

```
BRABZ <Xn>
```

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

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

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

Assembler Symbols

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

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

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

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

\begin{center}
\begin{tabular}{cccccccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \\
\end{tabular}
\end{center}

BRK \#<imm>

\begin{verbatim}
if HaveBTIExt() then
  SetBTypeCompatible(TRUE);
\end{verbatim}

\textbf{Assembler Symbols}

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

\textbf{Operation}

\texttt{AArch64.SoftwareBreakpoint(imm16)};
**BTI**

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

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

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

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

**System**
(Armv8.5)

```
1  1  0  1  0  1  0  0  |  1  0  1  1  0  0  1  0  0  1  0  0  1  0  0  1  0  x  x  0  1  1  1  1
|  0  1  0  |  1  0  1  0  0  |  0  1  0  |  1  0  0  |  1  1  1  1  1

CRm    op2
```

BTI `{<targets>}`

```markdown
SystemHintOp op;
```

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

**Assembler Symbols**

| `<targets>` | Is the type of indirection, encoded in “op2<2:1>”:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>op2&lt;2:1&gt;</code></td>
<td><code>&lt;targets&gt;</code></td>
</tr>
<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 
    when SystemHintOp_YIELD 
        Hint_Yield();
    when SystemHintOp_DGH 
        Hint_DGH();
    when SystemHintOp_WFE 
        Hint_WFE(-1);
    when SystemHintOp_WFI 
        Hint_WFI(-1);
    when SystemHintOp_SEV 
        SendEvent();
    when SystemHintOp_SEVL 
        SendEventLocal();
    when SystemHintOp_ESB 
        SynchronizeErrors();
        AArch64.vESBOperation(); 
        if PSTATE.EL IN {EL0, EL1} & EL2Enabled() then 
            TakeUnmaskedSErrorInterrupts();
    when SystemHintOp_PSB 
        ProfilingSynchronizationBarrier();
    when SystemHintOp_TSB 
        TraceSynchronizationBarrier();
    when SystemHintOp_CSB 
        ConsumptionOfSpeculativeDataBarrier();
    when SystemHintOp_BT
        SetBTypeNext('00');
    otherwise // do nothing

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CAS, CASA, CASAL, CASL

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

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

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

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

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

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

**No offset**

(Armv8.1)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | L  | 1  | Rs | o0 | 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;
b_bits(datasize) comparevalue;
b_bits(datasize) newvalue;
b_bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

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

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

X[s] = ZeroExtend(data, regrsize);
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.
- CASB and CASALB store to memory with release semantics.
- CASB has neither acquire nor release semantics.

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

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

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

**No offset**  
(Armv8.1)

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

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

- CASAB <Ws>, <Wt>, [<Xn|SP>{,#0}]
- CASALB <Ws>, <Wt>, [<Xn|SP>{,#0}]
- CASB <Ws>, <Wt>, [<Xn|SP>{,#0}]
- 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_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(8) comparevalue;
bits(8) newvalue;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

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

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

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

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

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

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

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

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

No offset
(Armv8.1)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 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 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

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

if !HaveAtomicExt() then UNDEFINED;

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

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

Assembler Symbols

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

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

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

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

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

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

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

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

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

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

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

**No offset**

(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | L  | 1  | Rs | o0 | 1  | 1  | 1  | 1  | 1  | Rn |     | Rt |     |     |     |     |     |     |     |     |     |     |

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

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

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

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

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

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

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

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

if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
integer datasize = 32 << UInt(sz);
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

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

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

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

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

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

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

<Xt> Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
<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(ldacctype) then s1:s2 else s2:s1;
newvalue = if BigEndian(stacctype) then t1:t2 else t2:t1;

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

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

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

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

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

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

**CBNZ <Wt>, <label>**

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

**CBNZ <Xt>, <label>**

```plaintext
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**

```plaintext
bits(datasize) operand1 = X[t];
if IsZero(operand1) == FALSE then
    BranchTo(PC[] + offset, BranchType_DIR);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

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

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

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
</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);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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.

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

32-bit (sf == 0)

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>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>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</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>
</tr>
<tr>
<td>Rn</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nzcv</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)

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

64-bit (sf == 1)

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

```plaintext
integer n = UInt(Rn);
integer m = UInt(Rm);
ingernal 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];
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.
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>sf</th>
<th>Rm</th>
<th>cond</th>
<th>Rn</th>
<th>nzcv</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
if ConditionHolds(cond) then
  operand2 = NOT(operand2);
  (neg, 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)

![Instruction code]

CFINV

if !HaveFlagManipulateExt() then UNDEFINED;

Operation

PSTATE.C = NOT(PSTATE.C);

Operational information

If PSTATE.DIT is 1:

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

This is an alias of SYS. This means:

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

**System**  
**Armv8.5**

\[
\begin{array}{ccccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 & 23 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\hline
L & op1 & CRn & CRm & op2 & Rt \\
\end{array}
\]

CFP RCTX, \(<Xt>\)

is equivalent to

SYS #3, C7, C3, #4, \(<Xt>\)

and is always the preferred disassembly.

**Assembler Symbols**

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

**Operation**

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

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

This is an alias of CSINC. This means:

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

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

32-bit (sf == 0)

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

is equivalent to

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

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

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

is equivalent to

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

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

- <Wd> is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <cond> is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

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

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

This is an alias of CSINV. This means:

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

32-bit (sf == 0)

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

is equivalent to

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

and is the preferred disassembly when \( Rn = Rm \).

64-bit (sf == 1)

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

is equivalent to

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

and is the preferred disassembly when \( Rn = Rm \).

Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- `<cond>` is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

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

Clear Exclusive clears the local monitor of the executing PE.

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

CLREX {#<imm>}

// CRm field is ignored

**Assembler Symbols**

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

**Operation**

```
ClearExclusiveLocal(ProcessorID());
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

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

Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.

32-bit (sf == 0)

\[ \text{CLS } <Wd>, <Wn> \]

64-bit (sf == 1)

\[ \text{CLS } <Xd>, <Xn> \]

integer d = \text{UInt}(Rd);
integer n = \text{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 = \text{CountLeadingSignBits}(operand1);
\(X[d] = result<\text{datasize}-1:0>\);

**Operational information**

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

Count Leading Zeros counts the number of binary zero bits before the first binary one bit in the value of the source register, and writes the result to the destination register.

\[
\begin{array}{cccccccccccccccccccccccccccc}
\hline
\text{sf} & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\text{op}

\begin{align*}
\text{Rn} & \\
\text{Rd} & \\
\end{align*}

32-bit (sf == 0)

\text{CLZ} <Wd>, <Wn>

64-bit (sf == 1)

\text{CLZ} <Xd>, <Xn>

\begin{align*}
\text{integer } d & = \text{UInt}(\text{Rd}) \\
\text{integer } n & = \text{UInt}(\text{Rn}) \\
\text{integer } \text{datasize} & = \text{if } \text{sf} == '1' \text{ then } 64 \text{ else } 32;
\end{align*}

Assembler Symbols

\begin{align*}
<Wd> & \text{ Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.} \\
<Wn> & \text{ Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.} \\
<Xd> & \text{ Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.} \\
<Xn> & \text{ Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.}
\end{align*}

Operation

\begin{align*}
\text{integer } \text{result}; \\
\text{bits(} \text{datasize} \text{) } \text{operand1} & = \text{X}[n]; \\
\text{result} & = \text{CountLeadingZeroBits(} \text{operand1}); \\
\text{X}[d] & = \text{result}<\text{datasize}-1:0>;
\end{align*}

Operational information

If \text{PSTATE.DIT} is 1:

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

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

This is an alias of ADDS (extended register). This means:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMN <Wn|WSP>, <Wm>{, <extend> {<amount>}}

is equivalent to

ADDS WZR, <Wn|WSP>, <Wm>{, <extend> {<amount>}}

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn|SP>, <R><m>{, <extend> {<amount>}}

is equivalent to

ADDS XZR, <Xn|SP>, <R><m>{, <extend> {<amount>}}

and is always the preferred disassembly.

Assembler Symbols

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

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

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

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

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>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”:
If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

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

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

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

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

Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (immediate). This means:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMN <Wn|WSP>, #<imm>{, <shift>}

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn|SP>, #<imm>{, <shift>}

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

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

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMN <Wn>, <Wm>{, <shift> #<amount>}

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn>, <Xm>{, <shift> #<amount>}

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

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

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

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

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

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

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

This is an alias of **SUBS (extended register)**. This means:

- The encodings in this description are named to match the encodings of **SUBS (extended register)**.
- The description of **SUBS (extended register)** gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 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 | 1 | 0 | 0 | 1 | Rm | option | imm3 | Rn | 1 | 1 | 1 | 1 | 1
```

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

```
CMP <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

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

```
CMP <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

**Assembler Symbols**

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

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

- `<m>` is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- `<extend>` For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:...
If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

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

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

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

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

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (immediate). This means:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

\[
\text{CMP } \langle Wn|WSP \rangle, \#\langle \text{imm} \rangle, \langle \text{shift} \rangle
\]

is equivalent to

\[
\text{SUBS } \text{WZR}, \langle Wn|WSP \rangle, \#\langle \text{imm} \rangle, \langle \text{shift} \rangle
\]

and is always the preferred disassembly.

64-bit (sf == 1)

\[
\text{CMP } \langle Xn|SP \rangle, \#\langle \text{imm} \rangle, \langle \text{shift} \rangle
\]

is equivalent to

\[
\text{SUBS } \text{XZR}, \langle Xn|SP \rangle, \#\langle \text{imm} \rangle, \langle \text{shift} \rangle
\]

and is always the preferred disassembly.

Assembler Symbols

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

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

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of **SUBS (shifted register)**. This means:

- The encodings in this description are named to match the encodings of **SUBS (shifted register)**.
- The description of **SUBS (shifted register)** gives the operational pseudocode for this instruction.

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

**CMP** `<Wn>, <Wm>{, <shift> #<amount>}`

is equivalent to

**SUBS WZR, <Wn>, <Wm> {, <shift> #<amount>}

and is always the preferred disassembly.

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

**CMP** `<Xn>, <Xm>{, <shift> #<amount>}`

is equivalent to

**SUBS XZR, <Xn>, <Xm> {, <shift> #<amount>}

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":
  
<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

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

**Operation**

The description of **SUBS (shifted register)** gives the operational pseudocode for this instruction.

**Operational information**

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

Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This is an alias of SUBPS. This means:

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

### Integer

(ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

\[ \text{Xd} \]

CMPP \text{<Xn|SP>, <Xm|SP>} is equivalent to

\[ \text{SUBPS XZR, <Xn|SP>, <Xm|SP>} \]

and is always the preferred disassembly.

### Assembler Symbols

- \text{<Xn|SP>} is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Xn” field.
- \text{<Xm|SP>} is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Xm” field.

### Operation

The description of SUBPS gives the operational pseudocode for this instruction.
CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG. This means:

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

32-bit (sf == 0)

CNEG <Wd>, <Wn>, <cond>

is equivalent to

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

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

CNEG <Xd>, <Xn>, <cond>

is equivalent to

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

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

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

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

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

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

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

Operation

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

Operational information

If PSTATE.DIT is 1:

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

Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions, based on information
gathered from earlier execution within a particular execution context, from allowing later speculative execution within
that context to be observable through side-channels.
For more information, see CPP RCTX, Cache Prefetch Prediction Restriction by Context.

This is an alias of SYS. This means:

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

System
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  |
| L  | op1| CRn| CRm| op2|

CPP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #7, <Xt>

and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of SYS gives the operational pseudocode for this instruction.
CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In Armv8-A, this is an **OPTIONAL** instruction, and in Armv8.1 it is mandatory for all implementations to implement it. `ID_AA64ISAR0_EL1`. CRC32 indicates whether this instruction is supported.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | sz | 0  | 1  | 0  | 0  | Rm | 0  | 1  | 0  | 0  | Rn | 5  | 4  | 3  | 2  | 1  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

CRC32B (sf == 0 && sz == 00)

CRC32B <Wd>, <Wn>, <Wm>

CRC32H (sf == 0 && sz == 01)

CRC32H <Wd>, <Wn>, <Wm>

CRC32W (sf == 0 && sz == 10)

CRC32W <Wd>, <Wn>, <Wm>

CRC32X (sf == 1 && sz == 11)

CRC32X <Wd>, <Wn>, <Xm>

if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);

Assembler Symbols

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

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

<Xm> Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.

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

Operation

bits(32) acc = X[n]; // accumulator
bits(size) val = X[m]; // input value
bits(32) poly = 0x04C11DB7<31:0>;
bits(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.
**CRC32CB, CRC32CH, CRC32CW, CRC32CX**

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In Armv8-A, this is an *OPTIONAL* instruction, and in Armv8.1 it is mandatory for all implementations to implement it. `ID_AA64ISAR0_EL1`. CRC32 indicates whether this instruction is supported.

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

**CRC32CB (sf == 0 && sz == 00)**

- **CRC32CB** `<Wd>`, `<Wn>`, `<Wm>`

**CRC32CH (sf == 0 && sz == 01)**

- **CRC32CH** `<Wd>`, `<Wn>`, `<Wm>`

**CRC32CW (sf == 0 && sz == 10)**

- **CRC32CW** `<Wd>`, `<Wn>`, `<Wm>`

**CRC32CX (sf == 1 && sz == 11)**

- **CRC32CX** `<Wd>`, `<Wn>`, `<Xm>`

    if !HaveCRCExt() then UNDEFINED;
    integer d = UInt(Rd);
    integer n = UInt(Rn);
    integer m = UInt(Rm);
    if sf == '1' && sz != '11' then UNDEFINED;
    if sf == '0' && sz == '11' then UNDEFINED;
    integer size = 8 << UInt(sz);

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
- `<Wm>` Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

**Operation**

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

bits(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
```

CRm op2

Operation

```
ConsumptionOfSpeculativeDataBarrier();
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register to the destination register.

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

CSEL <Wd>, <Wn>, <Wm>, <cond>

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

CSEL <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.

#### Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
if ConditionHolds (cond) then
    result = operand1;
else
    result = operand2;
X[d] = result;
```

#### Operational information

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

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

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

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

<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>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rm</td>
<td>cond</td>
<td>o2</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CSET `<Wd>`, `<cond>`

is equivalent to

`CSINC `<Wd>`, WZR, WZR, invert(<cond>)`

and is always the preferred disassembly.

64-bit (sf == 1)

CSET `<Xd>`, `<cond>`

is equivalent to

`CSINC `<Xd>`, XZR, XZR, invert(<cond>)`

and is always the preferred disassembly.

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<cond>` Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

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

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

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This is an alias of **CSINV**. This means:

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

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | != 1 | 1 | x | 0 | 0 | 1 | 1 | 1 | 1 | 1 | Rd |
| op |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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

CSETM <Wd>, <cond>

is equivalent to

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

and is always the preferred disassembly.

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

CSETM <Xd>, <cond>

is equivalent to

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

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<cond>` Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

**Operation**

The description of **CSINV** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

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

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

This instruction is used by the aliases CINC, and CSET.

32-bit (sf == 0)

CSINC <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSINC <Xd>, <Xn>, <Xm>, <cond>

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

Assembler Symbols

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

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINC</td>
<td>Rm != '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn != '11111' &amp;&amp; Rn == Rm</td>
</tr>
<tr>
<td>CSET</td>
<td>Rm == '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

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

if ConditionHolds(cond) then
    result = operand1;
else
    result = operand2 + 1;

X[d] = result;

Operational information

If PSTATE.DIT is 1:

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

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

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

This instruction is used by the aliases CINV, and CSETM.

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

32-bit (sf == 0)

CSINV <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSINV <Xd>, <Xn>, <Xm>, <cond>

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

Assembler Symbols

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

Alias Conditions

<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.
CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>cond</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CSNEG <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSNEG <Xd>, <Xn>, <Xm>, <cond>

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

Assembler Symbols

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

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNEG</td>
<td>cond != '111x' &amp;&amp; Rn == Rm</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
if ConditionHolds(cond) then
  result = operand1;
else
  result = NOT(operand2);
  result = result + 1;
X[d] = result;

Operational information

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Data Cache operation. For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 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>L</th>
<th>CRm</th>
<th>op1</th>
<th>CRn</th>
<th>op2</th>
<th>Rt</th>
</tr>
</thead>
</table>

DC <dc_op>, <Xt>
is equivalent to
SYS #<op1>, C7, <Cm>, #<op2>, <Xt>
and is the preferred disassembly when SysOp(op1,'0111',CRm,op2) == Sys_DC.

Assembler Symbols

<dc_op> Is a DC instruction name, as listed for the DC system instruction group, encoded in "op1:CRm:op2":

<table>
<thead>
<tr>
<th>op1 CRm op2</th>
<th>&lt;dc_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0110 001</td>
<td>IVAC</td>
<td>-</td>
</tr>
<tr>
<td>000 0110 010</td>
<td>ISW</td>
<td>-</td>
</tr>
<tr>
<td>000 0110 011</td>
<td>IGVAC</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>000 0110 100</td>
<td>IGSW</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>000 0110 101</td>
<td>IGDVAC</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>000 0110 110</td>
<td>IGDSW</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>000 1010 010</td>
<td>CSW</td>
<td>-</td>
</tr>
<tr>
<td>000 1010 100</td>
<td>CGSW</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>000 1010 110</td>
<td>CGDSW</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>000 1110 010</td>
<td>CISW</td>
<td>-</td>
</tr>
<tr>
<td>000 1110 100</td>
<td>CIGSW</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>000 1110 110</td>
<td>CIGDSW</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 0100 001</td>
<td>ZVA</td>
<td>-</td>
</tr>
<tr>
<td>011 0100 011</td>
<td>GVA</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 0100 100</td>
<td>GZVA</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1010 001</td>
<td>CVAC</td>
<td>-</td>
</tr>
<tr>
<td>011 1010 011</td>
<td>CGVAC</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1010 101</td>
<td>CGDVAC</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1011 001</td>
<td>CVAP</td>
<td>-</td>
</tr>
<tr>
<td>011 1100 001</td>
<td>CVADP</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1100 011</td>
<td>CVGAP</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1100 101</td>
<td>CGDVAP</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1101 001</td>
<td>CVADP</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1101 011</td>
<td>CGVADP</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1101 101</td>
<td>CGDVADP</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1110 001</td>
<td>CIVAC</td>
<td>-</td>
</tr>
<tr>
<td>011 1110 011</td>
<td>CIGVAC</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>011 1110 101</td>
<td>CIGDVAC</td>
<td>FEAT MTE</td>
</tr>
</tbody>
</table>

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
Debug Change PE State to EL1, when executed in Debug state:
  • If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
  • Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:
  • EL1 if the instruction is executed at EL0.
  • Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:
  • ELR_ELx becomes UNKNOWN.
  • SPSR_ELx becomes UNKNOWN.
  • ESR_ELx becomes UNKNOWN.
  • DLR_ELx and DSPSR_ELx become UNKNOWN.
  • The endianness is set according to SCTLR_ELx EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and HCR_EL2.TGE == 1.
This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see DCPS.

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 0 | 1 0 1 | imm16 | 0 0 0 | 0 1

DCPS1 {#<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);
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:
- \texttt{ELR \_ELx} becomes \texttt{UNKNOWN}.
- \texttt{SPSR \_ELx} becomes \texttt{UNKNOWN}.
- \texttt{ESR \_ELx} becomes \texttt{UNKNOWN}.
- \texttt{DLR \_EL0} and \texttt{DSPSR \_EL0} become \texttt{UNKNOWN}.
- The endianness is set according to \texttt{SCTLR \_ELx EE}.

This instruction is UNDEFINED at the following exception levels:
- All exception levels if EL2 is not implemented.
- At EL0 and EL1 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see \textit{DCPS}.

\begin{verbatim}
DCPS2 \{#<imm>\}
if \texttt{!Halted()} then UNDEFINED;
\end{verbatim}

\textbf{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.

\textbf{Operation}

\texttt{DCPSInstruction(LL);}

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:
• \textit{ELR EL3} becomes \textit{UNKNOWN}.
• \textit{SPSR EL3} becomes \textit{UNKNOWN}.
• \textit{ESR EL3} becomes \textit{UNKNOWN}.
• \textit{DLR EL0} and \textit{DSPSR EL0} become \textit{UNKNOWN}.
• The endianness is set according to \textit{SCTLR EL3}.EE.
This instruction is UNDEFINED at all exception levels if either:
• \textit{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 \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
\hline
1 1 0 1 0 1 0 0 | 1 0 1 | imm16 | 0 0 0 | 1 1 \hline
\end{verbatim}

\texttt{DCPS3 \{#<imm>\}}

\texttt{if !Halted() then UNDEFINED;}

\textbf{Assembler Symbols}

\texttt{<imm>} Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

\textbf{Operation}

\texttt{DCPSInstruction(LL)};

---

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

DGH is a hint instruction. A DGH instruction is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

**System**

(Armv8.6)

```plaintext
1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 1 1 1
```

DGH

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

**Operation**

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

```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 1 1 0 0 1 1 CRm | 1 0 1 1 1 1 1 1 |
| opc             |                 |                 |                 |
```

DMB `<option>`|`#<imm>`

```plaintext
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 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);

# <imm> syntax.

DMB
DRPS

Debug restore process state.

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.

It has encodings from 2 classes: Memory barrier and Memory nXS barrier.

### Memory barrier

```
31 30 29 28 27 26 25 24 23 22 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 | != 0x00 | 1 0 0 1 1 1 1 1
CRm   opc
```

**DSB <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 = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;

### Memory nXS barrier

(Armv8.7)

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

**DSB <option>nXS|#/imm>**

### Assembler Symbols

**<option>** For the memory barrier variant: specifies the limitation on the barrier operation. Values are:

- **SY**
  - Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

- **ST**
  - Full system is the required shareability domain, write is the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

- **LD**
  - Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

- **ISH**
  - Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

- **ISHST**
  - Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.
ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm, other than the values 0b0000 and 0b0100, that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

For the memory nXS barrier variant: specifies the limitation on the barrier operation. Values are:

**SY**
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm<3:2> = 0b11.

**ISH**
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b10.

**NSH**
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b01.

**OSH**
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b00.

<imm>
For the memory barrier variant: is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

For the memory nXS barrier variant: is a 5-bit unsigned immediate, encoded in “CRm<3:2>”:

<table>
<thead>
<tr>
<th>CRm&lt;3:2&gt;</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>16</td>
</tr>
<tr>
<td>01</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>28</td>
</tr>
</tbody>
</table>

**Operation**

DataSynchronizationBarrier(domain, types);
Data Value Prediction Restriction by Context prevents data value predictions, based on information gathered from earlier execution within an particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see DVP RCTX, Data Value Prediction Restriction by Context.

This is an alias of SYS. This means:

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

### System
(ARMv8.5)

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

DVP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #5, <Xt>

and is always the preferred disassembly.

**Assembler Symbols**

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

**Operation**

The description of SYS gives the operational pseudocode for this instruction.
EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

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

32-bit (sf == 0)

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

64-bit (sf == 1)

EON <Xd>, <Xn>, <Xm>{, <shift> #<amount>}</Xd>

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.
<br> 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 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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 0 1 0 0 1 0 0 N</td>
</tr>
<tr>
<td>opc</td>
</tr>
</tbody>
</table>

**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;
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 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</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>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)

EOR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}</p>

64-bit (sf == 1)

EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}</p>

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

\[
\begin{align*}
\text{bits(datasize)} & \quad \text{operand1} = X[n]; \\
\text{bits(datasize)} & \quad \text{operand2} = \text{ShiftReg}(m, \text{shift_type}, \text{shift_amount}); \\
\text{result} & \quad = \text{operand1} \text{ EOR operand2}; \\
X[d] & \quad = \text{result};
\end{align*}
\]

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.
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.

ERET

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

Operation

$AArch64.CheckForERetTrap(FALSE, TRUE);$;

bits(64) target = ELR[];

$AArch64.ExceptionReturn(target, SPSR[]);$
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)

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

#### 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  1  1  0  1  0  0  0  1  0  0  1  0  0  0  1  1  1  1  1
```

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

**Operation**

```
SynchronizeErrors();
AArch64_ESB0Operation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0Operation();
TakeUnmaskedSErrorInterrupts();
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Extract register extracts a register from a pair of registers.

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

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

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

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

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

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

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

if N != sf then UNDEFINED;
if sf == '0' & imms<5> == '1' then UNDEFINED;
lsb = UInt(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>ROR (immediate)</td>
<td>Rn == Rm</td>
</tr>
</tbody>
</table>

**Operation**

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(2*datasize) concat = operand1:operand2;
result = concat<lsb+datasize-1:lsb>;
X[d] = result;
Operational information

If PSTATE.DIT is 1:

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

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

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

Integer
(Armv8.5)

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

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

if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

Assembler Symbols

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

Operation

bits(64) address = if n == 31 then SP[] else X[n];
bits(64) mask = X[m];
bits(4) tag = AArch64.AllocationTagFromAddress(address);
mask<UInt(tag)> = '1';
X[d] = mask;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Hint instruction is for the instruction set space that is reserved for architectural hint instructions. Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

```
<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>
</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CRm</td>
<td>op2</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
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_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
  otherwise EndOfInstruction();
```

**Assembler Symbols**

<imm> is a 7-bit unsigned immediate, in the range 0 to 127 encoded in the "CRm:op2" field.

The encodings that are allocated to architectural hint functionality are described in the "Hints" table in the "Index by Encoding".

For allocated encodings of "CRm:op2":

HINT
• A disassembler will disassemble the allocated instruction, rather than the **HINT** instruction.
• An assembler may support assembly of allocated encodings using **HINT** with the corresponding `<imm>` value, but it is not required to do so.

**Operation**

```java
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_DGH
    Hint_DGH();
  when SystemHintOp_WFE
    Hint_WFE(-1);
  when SystemHintOp_WFI
    Hint_WFI(-1);
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64_ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0peration();
    TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    SetBTypeNext('00');
  otherwise    // do nothing
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Halt instruction. An HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

HLT

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0 0 1 0 imm16</td>
</tr>
</tbody>
</table>

HLT #<imm>

if EDS.CR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if HaveBTIExt() then
SetBTCompatible(TRUE);

Assembler Symbols

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

Operation

Hal(D); 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 \texttt{SCR\_EL3.HCE} is set to 0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in \texttt{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 1 0 0 0 0 1 0 0 0 1 0              | imm16          | 0 0 0 |
```

```
HVC #<imm>

// Empty.
```

**Assembler Symbols**

\(<imm>\) Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

**Operation**

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && (!IsSecure(EL2)Enabled() && 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);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Instruction Cache operation. For more information, see \textit{op0==0b01, cache maintenance, TLB maintenance, and address translation instructions}.

This is an alias of \texttt{SYS}. This means:

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

\begin{center}
\begin{tabular}{ccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & CRm & op2 & Rt & L & CRn \\
\end{tabular}
\end{center}

\texttt{IC \{, \ Xt\}}

is equivalent to

\texttt{SYS \#\{op1\}, C7, \{Cm\}, \{op2\}{, \ Xt\}}

and is the preferred disassembly when \texttt{SysOp(op1,'0111',CRm,op2) == Sys\_IC}.

\textbf{Assembler Symbols}

\texttt{<ic\_op>\{, \ Xt\}}

Is an IC instruction name, as listed for the IC system instruction pages, encoded in \texttt{"op1:CRm:op2"}:

\begin{center}
\begin{tabular}{cccccc}
\text{op1} & CRm & \text{op2} & <ic\_op> \\
000 & 0001 & 000 & IALLUIS \\
000 & 0101 & 000 & IALLU \\
011 & 0101 & 001 & IVAU \\
\end{tabular}
\end{center}

\texttt{<op1>}

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

\texttt{<Cm>}

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

\texttt{<op2>}

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

\texttt{<Xt>}

Is the 64-bit name of the optional general-purpose source register, defaulting to \texttt{\'11111\'}, encoded in the \texttt{"Rt"} field.

\textbf{Operation}

The description of \texttt{SYS} gives the operational pseudocode for this instruction.
Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

**Integer**

*(Armv8.5)*

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

IRG `<Xd|SP>, <Xn|SP>{, <Xm}>`

if `!HaveMTEExt()` then UNDEFINED;
integer d = `UInt(Xd)`;
integer n = `UInt(Xn)`;
integer m = `UInt(Xm)`;

**Assembler Symbols**

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

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

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

**Operation**

```
bits(64) operand = if n == 31 then SP[] else X[n];
bits(64) exclude_reg = X[m];
bits(16) exclude = exclude_reg<15:0> OR GCR_EL1.Exclude;

if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    if GCR_EL1.RRND == '1' then
        RGSR_EL1 = bits(64) UNKNOWN;
        if IsOnes(exclude) then
            rtag = '0000';
        else
            rtag = ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start = RGSR_EL1.TAG;
        bits(4) offset = AArch64.RandomTag();
        rtag = AArch64.ChooseNonExcludedTag(start, offset, exclude);
    else
        rtag = '0000';

bits(64) result = AArch64.AddressWithAllocationTag(operand, AccType_NORMAL, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```
ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.

```
<opc> ISB {<option>|#<imm>}
```

// No additional decoding required

**Assembler Symbols**

<option> Specifies an optional limitation on the barrier operation. Values are:

- **SY**
  Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

**Operation**

```
InstructionSynchronizationBarrier();
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD64B**

Single-copy Atomic 64-byte Load derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers, Xt to X(t+7). The data that is loaded is atomic and is required to be 64-byte aligned.

**Integer**

(Armv8.7)

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

LD64B <Xt>, [<Xn|SP> {,#0}]

if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

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

**Assembler Symbols**

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

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

**Operation**

CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemLoad64B(address, acctype);

for i = 0 to 7
    value = data<63+64*i:64*i>;
    if BigEndian(acctype) then value = BigEndianReverse(value);
    X[t+i] = value;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDADD, LDADDA, LDADDAL, LDDDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- LDADD has neither acquire nor release semantics.

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

This instruction is used by the alias **STADD, STADDL**.

**Integer**

*(Armv8.1)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 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>opc</th>
</tr>
</thead>
</table>
32-bit LDADD (size == 10 && A == 0 && R == 0)
LDADD <Ws>, <Wt>, [<Xn|SP>]

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

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

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

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

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

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

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

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_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>STADD, STADDL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDADD, LDADDA, LDADDAL, LDADDL
Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has neither acquire nor release semantics.

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

This instruction is used by the alias STADDB, STADDDL.

### 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 Rt |

- **size**
- **opc**

#### LDADDB (A == 1 && R == 0)

$$\text{LDADDB} <Ws>, <Wt>, [<Xn|SP>]$$

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

$$\text{LDADDAB} <Ws>, <Wt>, [<Xn|SP>]$$

#### LDADDB (A == 0 && R == 0)

$$\text{LDADDB} <Ws>, <Wt>, [<Xn|SP>]$$

#### LDADDLB (A == 0 && R == 1)

$$\text{LDADDLB} <Ws>, <Wt>, [<Xn|SP>]$$

if !HaveAtomicExt() then UNDEFINED;

- integer $t = \text{UInt}(\text{Rt})$;
- integer $n = \text{UInt}(\text{Rn})$;
- integer $s = \text{UInt}(\text{Rs})$;

```plaintext
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

- **<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, STADDDL</td>
<td>A == '0' &amp;&amp; Rt == '1111'</td>
</tr>
</tbody>
</table>
Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDADDH, LDADDAH, LDADDALH, LDADDLH**

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, **LDADDAH** and **LDADDALH** load from memory with acquire semantics.
- **LDADDLH** and **LDADDALH** store to memory with release semantics.
- **LDADDH** has neither acquire nor release semantics.

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

This instruction is used by the alias **STADDH, STADDLH**.

**Integer**
(Armv8.1)

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

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

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

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

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

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

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

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

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

if !HaveAtomicExt() then UNDEFINED;

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

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDH, STADDLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAignment();
    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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register. The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

**Integer (Armv8.3)**

<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>x</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>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 32-bit (size == 10) |

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

| 64-bit (size == 11) |

LDAPR <Xt>, [<Xn|SP> {},#0]

```plaintext
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**

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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 | 0 | 1 | 0 | 0 | 1 | (1) | (1) | (1) | (1) | (1) | 1 | 1 | 0 | 0 | 0 | 0 |   |   |   |   |   |   |   |   |   |   |

size Rs

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

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

Assembler Symbols

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

Operation

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

if HaveMTEExt() then
   SetTagCheckedInstruction(tag_checked);
if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n];
data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | Rn |   |
| size | Rs |

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**

```plaintext
bits(64) address;
bits(16) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType.ORDERED];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Unscaled offset
(Armv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | imm9 |

size opc

32-bit (size == 10)

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

64-bit (size == 11)

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

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

if n == 31 then
    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  | imm9         | 0  | 0  | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size | opc |

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

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

boolean tag_checked = n != 31;
```

Operation

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

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

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

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

**Unscaled offset**

(Armv8.4)

<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>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>
</tbody>
</table>

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

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

boolean tag_checked = n != 31;

**Operation**

if \texttt{HaveMTEExt}() then
  \texttt{SetTagCheckedInstruction}(tag_checked);

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{Mem}[address, 2, \texttt{AccType\_ORDERED}];
\texttt{X}[t] = \texttt{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  | 1  | 1  | 0  | 1  | 1  | x  | 0  |    | imm9 | 0  | 0  | Rn  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

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

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType.ORDERED] = data;
    when MemOp_LOAD
        data = Mem[address, 1, AccType.ORDERED];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURSH

Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Unscaled offset

(ARMv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 0  | 1  | 1  | x  | 0  | imm9| 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**size**
**opc**

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

```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 && (n != 31);
```
**Operation**

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

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

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[0];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 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>size</th>
<th>opc</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

LDAPURSW $\langle Xt \rangle$, $\langle Xn|SP\rangle\{, \#<simm>\}$

bits(64) offset = $\text{SignExtend}(imm9, 64)$;

**Assembler Symbols**

$\langle Xt \rangle$ is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

$\langle Xn|SP \rangle$ is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

$\langle simm \rangle$ 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);

boolean tag_checked = n != 31;
```

**Operation**

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

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

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

address = address + offset;

data = Mem[address, 4, AccType_ORDERED];
X[t] = SignExtend(data, 64);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes. For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

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

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

32-bit (size == 10)

\[ \text{LDAR} \ <Wt>, [<Xn|SP>\{,#0}\] \]

64-bit (size == 11)

\[ \text{LDAR} \ <Xt>, [<Xn|SP>\{,#0}\] \]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler Symbols

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

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

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

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

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

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

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(0)</th>
</tr>
</thead>
<tbody>
<tr>
<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>

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

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

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes. For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

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

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

bits(64) address;
bits(16) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

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

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

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

LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *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;

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

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
  // ConstrainedUNPREDICTABLE case
  X[t] = bits(datasize) UNKNOWN;  // In this case t = t2
elsif elsize == 32 then
  // 32-bit load exclusive pair (atomic)
  data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
  if BigEndian(AccType_ORDEREDATOMIC) then
    X[t] = data<datasize-1:elsize>;
    X[t2] = data<elsize-1:0>;
  else
    X[t] = data<elsize-1:0>;
    X[t2] = data<datasize-1:elsize>;
else // elsize == 64
  // 64-bit load exclusive pair (not atomic),
  // but must be 128-bit aligned
  if address != Align(address, dbytes) then
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_ORDEREDATOMIC, FALSE, FALSE));
  X[t] = Mem[address, 8, AccType_ORDEREDATOMIC];
  X[t2] = Mem[address+8, 8, AccType_ORDEREDATOMIC];

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

![LDAXR Opcode](image)

### 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

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ORDERED_ATOMIC];
X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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</td>
</tr>
</tbody>
</table>

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

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

Assembler Symbols

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

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

Operation

bits(64) address;
bits(8) data;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ORDEREDATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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.

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

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ORDERED_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.
LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDCLRA and LDCLRAL load from memory with acquire semantics.
- LDCLRL and LDCLRAL store to memory with release semantics.
- LDCLR has neither acquire nor release semantics.

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

This instruction is used by the alias STCLR, STCLRL.

**Integer**  
*(Armv8.1)*

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

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

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

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

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

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

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

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

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

if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Xt> Is the 64-bit name of the general-purpose register 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>
operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDCLR, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRB and LDCLRLB store to memory with release semantics.
- LDCLR has neither acquire nor release semantics.

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

This instruction is used by the alias STCLR, 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 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | A | R | 1 | Rs | 0 | 0 | 0 | 1 | 0 | 0 | Rn | Rt |

size opc

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

LDCLR <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>STCLR, STCLRLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
**Operation**

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH**

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, **LDCLRAH** and **LDCLRALH** load from memory with acquire semantics.
- **LDCLRLH** and **LDCLRALH** store to memory with release semantics.
- **LDCLRH** has neither acquire nor release semantics.

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

This instruction is used by the alias **STCLRH, STCLRLH**.

### Integer
*(Armv8.1)*

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

- **size**
- **opc**

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

**LDCLRAH** `<Ws >, <Wt >, [<Xn|SP>]`

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

**LDCLRALH** `<Ws >, <Wt >, [<Xn|SP>]`

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

**LDCLRH** `<Ws >, <Wt >, [<Xn|SP>]`

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

**LDCLRLH** `<Ws >, <Wt >, [<Xn|SP>]`

if !HaveAtomicExt() then UNDEFINED;

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

```plaintext
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

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

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);
if t != 31 then
  X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDEORA and LDEORAL load from memory with acquire semantics.
- LDEORL and LDEORAL store to memory with release semantics.
- LDEOR has neither acquire nor release semantics.

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

This instruction is used by the alias STEOR, STEORL.

Integer
(Armv8.1)

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

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

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

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

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

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

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

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

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

if ![HaveAtomicExt]() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEOR, STEORL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

\[
\begin{align*}
\text{bits(64) address;} \\
\text{bits(datasize) value;} \\
\text{bits(datasize) data;} \\
\text{if HaveMTEEExt() then} \\
\quad \text{SetTagCheckedInstruction(tag_checked);} \\
\text{value = X[s];} \\
\text{if n == 31 then} \\
\quad \text{CheckSPAlignment();} \\
\quad \text{address = SP[;]} \\
\text{else} \\
\quad \text{address = X[n];} \\
\text{data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);} \\
\text{if t != 31 then} \\
\quad \text{X[t] = ZeroExtend(data, regsize);} \\
\end{align*}
\]

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has neither acquire nor release semantics.

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

This instruction is used by the alias STEORB, STEORLB.

### Integer

**(Armv8.1)**

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

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

Page 197
Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**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 neither acquire nor release semantics.

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

This instruction is used by the alias **STEORH, STEORLH**.

### Integer

(Armv8.1)

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

- **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' & & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
  AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
  boolean tag_checked = n != 31;

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORH, STEORLH</td>
<td>A == '0' &amp; &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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)

<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>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>0</td>
<td>1</td>
<td>1</td>
<td>imm9</td>
<td>0</td>
<td>0</td>
<td>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

if !HaveMTEExt() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

bits(64) address;
bits(4) tag;

if n == 31 then
  CheckSAPAlignment();
  address = SP[];
else
  address = X[n];
address = address + offset;
address = Align(address, TAG_GRANULE);
tag = AArch64.MemTag[address, AccType_NORMAL];
X[t] = AArch64.AddressWithAllocationTag(X[t], AccType_NORMAL, tag);
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_AA64FPRI_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  | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

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

if ![HaveMTE2Ext]() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

### Assembler Symbols

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

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

### Operation

if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (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;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

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

### No offset
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | x  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | (1) | (1) | (1) | (1) | 0  | (1) | (1) | (1) | (1) | Rn  |   |   |   |   |   |   |   |   |
| size | L | Rs | o0 | Rt2 |

#### 32-bit (size == 10)

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

#### 64-bit (size == 11)

LDLR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

### Assembler Symbols

- **<Wt>** Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xt>** Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSAPAlignment();
  address = SP[];
else
  address = X[n];
data = Mem[address, dbytes, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, regsize);

#### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes. For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset
(Armv8.1)

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

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

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

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = Mem[address, 1, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

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

### No offset
*(Armv8.1)*

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

LDLARH `<Wt>`, `<Xn|SP>{,#0}`

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

boolean tag_checked = n != 31;

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = Mem[address, 2, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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.

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;

boolean rt_unknown = FALSE;

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

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

address = address + offset;

data1 = Mem[address, dbytes, AccType_STREAM];
data2 = Mem[address+dbytes, dbytes, AccType_STREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
data2 = bits(datasize) UNKNOWN;
X[t] = data1;
X[t2] = data2;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

**Post-index**

<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>
<th><em>x</em></th>
</tr>
</thead>
<tbody>
<tr>
<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>imm7</td>
<td>Rt2</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

<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>
<th><em>x</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm7</td>
<td>Rt2</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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

<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>
<th><em>x</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>imm7</td>
<td>Rt2</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<\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 <\texttt{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 <\texttt{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 <\texttt{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 <\texttt{imm}>/8.

Shared Decode

\begin{verbatim}
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;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF 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();
\end{verbatim}
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;

constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

if !postindex then
    address = address + offset;

data1 = Mem[address, dbytes, AccType_NORMAL];
data2 = Mem[address+dbytes, dbytes, AccType_NORMAL];

if rt unknown then
    data1 = bits(datasize) UNKNOWN;
data2 = bits(datasize) UNKNOWN;

if signed then
    X[t] = SignExtend(data1, 64);
    X[t2] = SignExtend(data2, 64);
else
    X[t] = data1;
    X[t2] = data2;

if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elseif 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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

```
0 1 1 0 1 0 0 1 1   imm7     Rt2     Rn     Rt
opc   L
```

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```java
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

```
0 1 1 0 1 0 0 1 1   imm7     Rt2     Rn     Rt
opc   L
```

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```java
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

```
0 1 1 0 1 0 0 1 0   imm7     Rt2     Rn     Rt
opc   L
```

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm}>]
```

```java
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDPSW*.

**Assembler Symbols**

- `<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as `<imm>/4`. For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as `<imm>/4`. 
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bits(64) offset = LSL(SignExtend(imm7, 64), 2);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
  when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
  when Constraint_UNKNOWN wb_unknown = TRUE;    // writeback is UNKNOWN
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP EndOfInstruction();
if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
  when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP EndOfInstruction();

Operation

bits(64) address;
bits(32) data1;
bits(32) data2;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
if !postindex then
  address = address + offset;
data1 = Mem[address, 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 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 (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

**Post-index**

```
|   31 |  30 |  29 |  28 |  27 |  26 |  25 |  24 |  23 |  22 |  21 |  20 |  19 |  18 |  17 |  16 |  15 |  14 |  13 |  12 |  11 |  10 |   9 |   8 |   7 |   6 |   5 |   4 |   3 |   2 |   1 |   0 |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 x  | 1   | 1   | 1   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| size  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| imm9  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 0 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Rn    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Rt    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
```

32-bit (size == 10)

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

64-bit (size == 11)

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

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

**Pre-index**

```
|   31 |  30 |  29 |  28 |  27 |  26 |  25 |  24 |  23 |  22 |  21 |  20 |  19 |  18 |  17 |  16 |  15 |  14 |  13 |  12 |  11 |  10 |   9 |   8 |   7 |   6 |   5 |   4 |   3 |   2 |   1 |   0 |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 x  | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| size  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| imm9  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 1 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Rn    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Rt    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
```

32-bit (size == 10)

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

64-bit (size == 11)

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

```java
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

**Unsigned offset**

```
|   31 |  30 |  29 |  28 |  27 |  26 |  25 |  24 |  23 |  22 |  21 |  20 |  19 |  18 |  17 |  16 |  15 |  14 |  13 |  12 |  11 |  10 |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 x  | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| size  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| imm12 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Rn    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Rt    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
```

Unsigned offset
**32-bit (size == 10)**

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

**64-bit (size == 11)**

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

```plaintext
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bbox(64) offset = LSL(ZeroExtend(imm12, 64), scale);

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

**Assembler Symbols**

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

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

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

<simm> For the 32-bit variant: is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

**Shared Decode**

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

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;
boolean wb_unknown = FALSE;

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

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

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elseif 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.

<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>1</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>imm19</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

LDR <Wt>, <label>

64-bit (opc == 01)

LDR <Xt>, <label>

integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;
offset = SignExtend(imm19:'00', 64);

Assembler Symbols

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

Operation

bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(FALSE);

case memop of
  when MemOp_LOAD
    data = Mem[address, size, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, 64);
    else
      X[t] = data;
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see Load/Store addressing modes.

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
    SetTagCheckedInstruction(TRUE);

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

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

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for LDRAA, and key B is used for LDRAB.

If the authentication passes, the PE behaves the same as for an LDR instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

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

Unscaled offset
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 1  | 0  | 0  | 0  | M  | S  | 1  | imm9 | W  | 1  | Rn  |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size

Key A, offset (M == 0 & & W == 0)

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

Key A, pre-indexed (M == 0 & & W == 1)

LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!

Key B, offset (M == 1 & & W == 0)

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

Key B, pre-indexed (M == 1 & & W == 1)

LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!

if !HavePACExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bis(10) S10 = S:imm9;
bis(64) offset = LSL(SignExtend(S10, 64), 3);
boolean tag_checked = wback | | n != 31;

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.
Operation

bits(64) address;
bits(64) data;
boolean wb_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

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

Pre-index

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

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

Unsigned offset

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

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 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.
<imm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = wback || n != 31;
boolean wb_unknown = FALSE;

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

Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

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

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

if !postindex then
  address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
  if wb unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

Extended register (option != 011)

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
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
    SetTagCheckedInstruction(TRUE);

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

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

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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

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

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

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

Pre-index

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

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

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

Unsigned offset

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

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

boolean wback = FALSE;
boolean postindex = FALSE;
bhits(64) offset = LSL(ZeroExtend(imm12, 64), 1);

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

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.
Shared Decode

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

boolean tag_checked = wback || \( n \neq 31; \)

boolean wb_unknown = FALSE;

if wback && \( n == t \) && \( n \neq 31 \) then
\[
c = \text{ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);}\]
assert c \( \in \{ \text{Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP}\}; \)

\text{case} c \text{ of}
\[
\begin{align*}
\text{when Constraint_WBSUPPRESS} & \quad \text{wback} = \text{FALSE}; & \quad \text{writeback is suppressed} \\
\text{when Constraint_UNKNOWN} & \quad \text{wb unknown} = \text{TRUE}; & \quad \text{writeback is UNKNOWN} \\
\text{when Constraint_UNDEF} & \quad \text{UNDEFINED}; \\
\text{when Constraint_NOP} & \quad \text{EndOfInstruction}();
\end{align*}
\]

Operation

if \( \text{HaveMTEExt}() \) then
\[
\text{SetTagCheckedInstruction}(\text{tag_checked});
\]

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

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

if !\text{postindex} then
\[
\text{address} = \text{address} + \text{offset};
\]

data = \text{Mem}[\text{address}, 2, \text{AccType_NORMAL}];
\[
\text{X}[t] = \text{ZeroExtend}(\text{data}, 32);
\]

if wback then
\[
\text{if wb unknown then}
\quad \text{address} = \text{bits(64) UNKNOWN};
\quad \text{elsif postindex then}
\quad \text{address} = \text{address} + \text{offset};
\quad \text{if } n == 31 \text{ then}
\quad \text{SP[]} = \text{address};
\quad \text{else}
\quad \text{X}[n] = \text{address};
\]

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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*.

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

size opc

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
    SetTagCheckedInstruction(TRUE);

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 1 0 0 0 1 x 0 | imm9 | 0 1 | Rn | Rt |
| size | opc |

32-bit (opc == 11)

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

64-bit (opc == 10)

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

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

**Pre-index**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 0 1 1 1 0 0 0 1 x 0 | imm9 | 1 1 | Rn | Rt |
| size | opc |

32-bit (opc == 11)

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

64-bit (opc == 10)

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

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

**Unsigned offset**

| 31 30 29 28 27 26 25 24 23 22 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 x | imm12 | Rn | Rt |
| size | opc |
32-bit (opc == 11)

LDRSB \(<Wt>\), \([<Xn|SP>{{, #<pimm>}}]\)

64-bit (opc == 10)

LDRSB \(<Xt>\), \([<Xn|SP>{{, #<pimm>}}]\)

boolean wback = FALSE;
boolean postindex = FALSE;
bhits(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.
\(<\text{simm}>\) Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
\(<\text{pimm}>\) Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_WBSUPPRESS wback = FALSE;  // writeback is suppressed
      when Constraint_UNKNOWN wb unknown = TRUE;  // writeback is UNKNOWN
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_NONE rt unknown = FALSE;   // value stored is original value
      when Constraint_UNKNOWN rt unknown = TRUE; // value stored is UNKNOWN
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];
endif

if !postindex then
    address = address + offset;
endif

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t];
            Mem[address, 1, AccType_NORMAL] = data;
        endif
    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
        endif
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
endif

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    endif
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
    endif
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.
LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | x | 1 | Rm | option | S | 1 | 0 | Rn | Rt |

**32-bit with extended register offset (opc == 11 & option != 011)**

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

**32-bit with shifted register offset (opc == 11 & option == 011)**

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

**64-bit with extended register offset (opc == 10 & option != 011)**

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

**64-bit with shifted register offset (opc == 10 & option == 011)**

LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  

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

ExtendType extend_type = DecodeRegExtend(option);

**Assembler Symbols**

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

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

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

- **<Wm>**  
  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
    SetTagCheckedInstruction(tag_checked);

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

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];
else
    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 |

**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 |

**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 |

LDRSH (immediate)
32-bit (opc == 11)

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

64-bit (opc == 10)

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

boolean wback = FALSE;
boolean postindex = FALSE;
b_bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);

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

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_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();

LDRSH (immediate)
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[1];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(16) UNKNOWN;
        else
            data = X[t];
            Mem[address, 2, AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see Load/Store addressing modes.

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

32-bit (opc == 11)

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

64-bit (opc == 10)

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

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

ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
bits(64) address;
bits(16) data;
if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];
address = address + offset;
case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 2, AccType_NORMAL] = data;
  when MemOp_LOAD
    data = Mem[address, 2, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);
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

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

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

```java
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>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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>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>
</tr>
<tr>
<td>size</td>
<td>opc</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```java
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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>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>
</tr>
<tr>
<td>size</td>
<td>opc</td>
<td>imm12</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```java
boolean wback = FALSE;
boolean postindex = FALSE;
bite(64) offset = LSL(ZeroExtend(imm12, 64), 2);  
```

For information about the constrained unpredictable behavior of this instruction, see *Architectural Constraints on Unpredictable behaviors*, and particularly *LDRSW (immediate)*.

### Assembler Symbols

- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>` Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as `<pimm>/4`. 
Shared Decode

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

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

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

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

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

if !postindex then
    address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    else 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.
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.

Assembler Symbols

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

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

Operation

```
bits(64) address = PC[] + offset;
bits(32) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);
    data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see Load/Store addressing modes.

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

size opc

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

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

ExtendType extend_type = DecodeRegExtend(option);

type shift = if S == '1' then 2 else 0;

Assembler Symbols

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

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

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTW</td>
</tr>
<tr>
<td>001</td>
<td>LSL</td>
</tr>
<tr>
<td>010</td>
<td>SXTW</td>
</tr>
<tr>
<td>011</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

Shared Decode

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

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

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

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

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDSET, LDSETA, LDSETAL, LDSETL**

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- LDSET has neither acquire nor release semantics.

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

This instruction is used by the alias **STSET, STSETL**.

**Integer**

(ARMv8.1)

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

_This instruction is used by the alias **STSET, STSETL**._

LDSET, LDSETA, LDSETAL, LDSETL
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;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp\_ORR, value, ldacctype, stacctype);

if t != 31 then
    \text{X}[t] = \text{ZeroExtend}(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has neither acquire nor release semantics.

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

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

This instruction is used by the alias STSETB, STSETLB.

### Integer

(Armv8.1)

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

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

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

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

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

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

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

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

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

if !HaveAtomicExt() then UNDEFINED;

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

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETB, STSETLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDSETH, 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 neither acquire nor release semantics.

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

This instruction is used by the alias STSETH, STSETLH.

**Integer**

(Armv8.1)

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

**Op Code (opc)**

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

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

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

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

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

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

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

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

if !HaveAtomicExt() then UNDEFINED;

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

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

**Assembler Symbols**

- `<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETH, STSETLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

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

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAXL and LDSMAXAL store to memory with release semantics.
- LDSMAX has neither acquire nor release semantics.

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

This instruction is used by the alias STSMAX, STSMAXL.

Integer (Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 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_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>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
    SetTagCheckedInstruction(tag_checked);

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

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has neither acquire nor release semantics.

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

This instruction is used by the alias STSMAXB, STSMAXLB.

Integer
(Armv8.1)

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

| size | opc |

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

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

LDSMAXALB (A == 1 && R == 1)

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

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

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

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

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

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt>  Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXB, STSMAXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMAXH, STSMAXLH.

### Integer (Armv8.1)

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 0</td>
<td>A R 1</td>
</tr>
</tbody>
</table>

**LDSMAXAH (A == 1 && R == 0)**

LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXALH (A == 1 && R == 1)**

LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXH (A == 0 && R == 0)**

LDSMAXH <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXLH (A == 0 && R == 1)**

LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

- `<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXH_STSMAXLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDSMIN, LDSMINA, LDSMINAL, LDSMINL**

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, **LDSMINA** and **LDSMINAL** load from memory with acquire semantics.
- **LDSMINL** and **LDSMINAL** store to memory with release semantics.
- **LDSMIN** has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.
For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STSMIN, STSMINL**.

### Integer

**(Armv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**size** | **opc**

Page 262
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_ORDERED_ATOMIC_RW else AccType_ATOMIC_RW;
AccType statcctype = if R == '1' then AccType_ORDERED_ATOMIC_RW else AccType_ATOMIC_RW;
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
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
  X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*
For information about memory accesses see *Load/Store addressing modes.*

This instruction is used by the alias STSMINB, STSMINLB.

**Integer (Armv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | A  | 1  | R  | 0  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn |     |     |     |     |     |

**LDSMINB (A == 1 && R == 0)**

LDSMINB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINALB (A == 1 && R == 1)**

LDSMINALB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINB (A == 0 && R == 0)**

LDSMINB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINLB (A == 0 && R == 1)**

LDSMINLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMINB, STSMINLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment()
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMINH, STSMINLH.

### Integer
**Armv8.1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn | Rt |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>

### 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>]

```assembly
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
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | imm9| 1  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size| opc|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (size == 10)

LDTR &lt;Wt&gt;, [&lt;Xn|SP&gt;{, #&lt;simm&gt;}]}

64-bit (size == 11)

LDTR &lt;Xt&gt;, [&lt;Xn|SP&gt;{, #&lt;simm&gt;}]

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

&lt;Wt&gt; Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
&lt;Xt&gt; Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
&lt;Xn|SP&gt; Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
&lt;simm&gt; 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 &amp;&amp; !(EL2Enabled() &amp;&amp; HaveNVExt() &amp;&amp; HCR_EL2.&lt;NV,NV1&gt; == '11');
unpriv_at_el2 = PSTATE.EL == EL2 &amp;&amp; HaveVirtHostExt() &amp;&amp; HCR_EL2.&lt;E2H,TGE&gt; == '11';

user_access_override = HaveUAOExt() &amp;&amp; PSTATE.UAO == '1';
if !user_access_override &amp;&amp; (unpriv_at_el1 || unpriv_at_el2) then
  accctype = AccType_UNPRIV;
else
  accctype = AccType_NORMAL;

integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 &lt;&lt; scale;
boolean tag_checked = n != 31;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, acctype];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.<E2H, TGE> is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

```
0 0 1 1 0 0 0 1 0
imm9
1 0
Rn
Rt

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
    accype = AccType_UNPRIV;
else
    accype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

**Operation**

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data = Mem[address, 1, accype];
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.

![Address Calculation](https://via.placeholder.com/150)

| 31 30 29 28 27 26 25 24 23 22 21 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 | imm9            | 1 0             | Rn              | Rt              |

size opc

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

```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
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, acctype];
X[t] = ZeroExtend(data, 32);
```

LDTRH
**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.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| size             | opc              | imm9            | 1 0  Rn | Rt |

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.
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && 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 
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
if n == 31 then 
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else 
  address = X[n];
address = address + offset;
case memop of 
  when MemOp_STORE 
    data = X[t];
    Mem[address, 1, acctype] = 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

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;

MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);

Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];
address = address + offset;
case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 2, acctype] = data;
  when MemOp_LOAD
    data = Mem[address, 2, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.E2H, TGE is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>1 1 1 0 0 0</td>
</tr>
</tbody>
</table>

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() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;
boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(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 neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STUMAX, STUMAXL.

**Integer (Armv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 0  | 0  | Rn | 0  | 1  | 1  | 0  | 0  | Rn | 0  | 1  | 1  | 0  | 0  | Rn |

This table provides the format for the LDUMAX instruction, with fields for size, opc, and the registers Rs, Rn, and Rt.
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 datatype = 8 << UInt(size);
    integer regsize = if datatype == 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
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);
if t != 31 then
  X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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.
- `LDUMAXB` and `LDUMAXALB` store to memory with release semantics.
- `LDUMAXLB` has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias `STUMAXB, STUMAXLB`.

### Integer (Armv8.1)

![Integer Table](image)

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Wt>` is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXB_STUMAXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMAXH, STUMAXLH.

### Integer

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A | R | 1  | Rs | 0  | 1  | 1  | 0  | 0  | Rn | Rt |

**size** | **opc**

### LDUMAXAH (A == 1 && R == 0)

LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]

### LDUMAXALH (A == 1 && R == 1)

LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]

### LDUMAXH (A == 0 && R == 0)

LDUMAXH <Ws>, <Wt>, [<Xn|SP>]

### LDUMAXLH (A == 0 && R == 1)

LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
in integer n = UInt(Rn);
in 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

<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
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDUMIN, LDUMINA, LDUMINAL, LDUMINL**

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMINA and LDUMINAL load from memory with acquire semantics.
- LDUMINL and LDUMINAL store to memory with release semantics.
- LDUMIN has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*

For information about memory accesses see *Load/Store addressing modes.*

This instruction is used by the alias `STUMIN, STUMINL`.

**Integer**

(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn | R  | 1  |

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 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>STUMIN, STUMINL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
b改制 (datasize) value;
b改制 (datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
- LDUMINLB and LDUMINALB store to memory with release semantics.
- LDUMINB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMINB, STUMINLB.

### Integer

**Armv8.1**

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0  0 1  1 1 0  0  0  A R 1 Rs  0  1 1  1 0  0  Rn  Rt
```

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<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>]

```java
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' &amp; Rt != '11111' then AccType_ORDERED_ATOMIC_RW else AccType_ATOMIC_RW;
AccType stacctype = if R == '1' then AccType_ORDERED_ATOMIC_RW else AccType_ATOMIC_RW;
boolean tag_checked = n != 31;
```

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINB, STUMINLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMINH, STUMINLH.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn | 0  | 0  | Rt |

size opc

LDUMINH (A == 1 && R == 0)

LDUMINAH <Ws>, <Wt>, [<Xn|SP>]

LDUMINALH (A == 1 && R == 1)

LDUMINAH <Ws>, <Wt>, [<Xn|SP>]

LDUMINH (A == 0 && R == 0)

LDUMINH <Ws>, <Wt>, [<Xn|SP>]

LDUMINLH (A == 0 && R == 1)

LDUMINLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '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>STUMINH, STUMINLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
**Operation**

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
  X[t] = ZeroExtend(data, 32);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDUR**

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

```
|   31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  23  |  22  |  21  |  20  |  19  |  18  |  17  |  16  |  15  |  14  |  13  |  12  |  11  |  10  |  9   |  8   |  7   |  6   |  5   |  4   |  3   |  2   |  1   |  0   |
|-------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| size  | opc  | imm9 | Rn   | Rt   |
```

### 32-bit (size == 10)

```
LDUR <Wt>, [<Xn|SP>{, #<simm>}]  
```

### 64-bit (size == 11)

```
LDUR <Xt>, [<Xn|SP>{, #<simm>}]  
```

- integer scale = UInt(size);
- bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```
integer n  = UInt(Rn);
integer t  = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

**Operation**

```
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 0 1 1 0 0 0 0 1 0 | imm9 | 0 0 | Rn | Rt |
```

size  opc

LDURB <Wt>, [<Xn|SP>{, #<simm>}

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = 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.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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</td>
</tr>
<tr>
<td>0 1 1 1 0 0 0 0 1 0  imm9</td>
</tr>
</tbody>
</table>
```

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

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

Operation

```java
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(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.

|   |   |   |   |   |   |   | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | x | 0 | imm9 | 0 | 0 | Rn | Rt |
| size | opc |

32-bit (opc == 11)

LDURSB <Wt>, [<Xn|SP>{, #<simm>}]

64-bit (opc == 10)

LDURSB <Xt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP{};
else
  address = X[n];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 1, AccType_NORMAL] = data;

  when MemOp_LOAD
    data = Mem[address, 1, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 (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 HaveMTExeExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 2, AccType_NORMAL] = data;

  when MemOp_LOAD
    data = Mem[address, 2, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCH
    Forfetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

size  opc
```

LDURSW <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

```java
type integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```java
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDXP**

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | sz | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |
| L  | Rs | 0  | 0  |

**32-bit (sz == 0)**

LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

**64-bit (sz == 1)**

LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then

    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE;   // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDXP*.

**Assembler Symbols**

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
b bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;  // In this case t = t2
elseif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ATOMIC];
    if BigEndian(AccType_ATOMIC) then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else
    // 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
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ATOMIC];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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.

31 30 29 28 27 26 25 24 23 22 21 20 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 1 (1) (1) (1) (1) 0 (1) (1) (1) (1) Rn  Rt

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
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_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 \textit{Synchronization and semaphores}. For information about memory accesses see \textit{Load/Store addressing modes}.

\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 1 0 0 0 0 1 0 (1)(1)(1)(1)(1) 0 (1)(1)(1)(1)(1)
size L Rs o0 Rt2

LDXRH <Wt>, [<Xn|SP>{,0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
\end{verbatim}

\textbf{Assembler Symbols}

\begin{itemize}
  \item <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
  \item <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\end{itemize}

\textbf{Operation}

\begin{verbatim}
bits(64) address;
bits(16) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);
\end{verbatim}

\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.
LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This is an alias of LSLV. This means:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode for this instruction.

32-bit (sf == 0)

LSL <Wd>, <Wn>, <Wm>

is equivalent to

LSLV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

LSL <Xd>, <Xn>, <Xm>

is equivalent to

LSLV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSLV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | N  | immr| != x11111 | Rn | Rd |
| opc |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |     |
```

32-bit (sf == 0 && N == 0 && imms != 01111)

LSL <Wd>, <Wn>, #<shift>

is equivalent to

UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)

and is the preferred disassembly when `imms + 1 == immr`.

64-bit (sf == 1 && N == 1 && imms != 11111)

LSL <Xd>, <Xn>, #<shift>

is equivalent to

UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)

and is the preferred disassembly when `imms + 1 == immr`.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<shift>` For the 32-bit variant: is the shift amount, in the range 0 to 31.
  For the 64-bit variant: is the shift amount, in the range 0 to 63.

**Operation**

The description of UBFM gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSLV

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

```text
LSLV <Wd>, <Wn>, <Wm>
```

**64-bit (sf == 1)**

```text
LSLV <Xd>, <Xn>, <Xm>
```

```text
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**

```text
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.

---

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of **LSRV**. This means:

- The encodings in this description are named to match the encodings of **LSRV**.
- The description of **LSRV** gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf | 0 0 1 1 0 1 0 1 1 0 | Rm | 0 0 1 0 0 1 | Rn | Rd
```

**32-bit (sf == 0)**

```
LSR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSRV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

**64-bit (sf == 1)**

```
LSR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSRV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

The description of **LSRV** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**LSR (immediate)**

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010110 N</td>
<td>111111</td>
<td>111111</td>
</tr>
<tr>
<td>immr</td>
<td>x 1 1 1 1 1</td>
<td>Rn</td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0 && N == 0 && imms == 011111)**

LSR \(<Wd>, <Wn>, #<shift>\)

is equivalent to

UBFM \(<Wd>, <Wn>, #<shift>, #31\)

and is always the preferred disassembly.

**64-bit (sf == 1 && N == 1 && imms == 111111)**

LSR \(<Xd>, <Xn>, #<shift>\)

is equivalent to

UBFM \(<Xd>, <Xn>, #<shift>, #63\)

and is always the preferred disassembly.

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<shift>** For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

**Operation**

The description of UBFM gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias **LSR (register)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

**LSRV** `<Wd>`, `<Wn>`, `<Wm>`

**64-bit (sf == 1)**

**LSRV** `<Xd>`, `<Xn>`, `<Xm>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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 0 1 1 0 1 1 0 0 0</th>
<th>Rm</th>
<th>0</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></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));
Xd = result<destsize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode for this instruction.

<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>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</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>
</tr>
</tbody>
</table>

### 32-bit (sf == 0)

MNEG <Wd>, <Wn>, <Wm>

is equivalent to

MSUB <Wd>, <Wn>, <Wm>, WZR

and is always the preferred disassembly.

### 64-bit (sf == 1)

MNEG <Xd>, <Xn>, <Xm>

is equivalent to

MSUB <Xd>, <Xn>, <Xm>, XZR

and is always the preferred disassembly.

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of MSUB gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (to/from SP)

Move between register and stack pointer: Rd = Rn.

This is an alias of ADD (immediate). This means:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

MOV <Wd|WSP>, <Wn|WSP>

is equivalent to

ADD <Wd|WSP>, <Wn|WSP>, #0

and is the preferred disassembly when (Rd == '11111' || Rn == '11111').

64-bit (sf == 1)

MOV <Xd|SP>, <Xn|SP>

is equivalent to

ADD <Xd|SP>, <Xn|SP>, #0

and is the preferred disassembly when (Rd == '11111' || Rn == '11111').

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.
MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This is an alias of MOVN. This means:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode for this instruction.

32-bit (sf == 0 & hw == 0x)

MOV <Wd>, #<imm>

is equivalent to

MOVN <Wd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) & hw != '00') & ! IsOnes(imm16).

64-bit (sf == 1)

MOV <Xd>, #<imm>

is equivalent to

MOVN <Xd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) & hw != '00')

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm> For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff

For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".

<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.

32-bit (sf == 0 && hw == 0x)

MOV <Wd>, #<imm>

is equivalent to

MOVZ <Wd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when !(IsZero(imm16) && hw != '00').

64-bit (sf == 1)

MOV <Xd>, #<imm>

is equivalent to

MOVZ <Xd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when !(IsZero(imm16) && hw != '00').

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<imm> For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw".

For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".

<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 1  | 1  | 0  | 0  | 0  | 0  | N  | immr | imms | 1  | 1  | 1  | 1  | 1  | Rd |
| opc|    |    |    |    |    |    |    |    |    |      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit \((sf == 0 && N == 0)\)

MOV <Wd|WSP>, #<imm>

is equivalent to

ORR <Wd|WSP>, WZR, #<imm>

and is the preferred disassembly when \(!\text{MoveWidePreferred}(sf, N, imms, immr)\).

64-bit \((sf == 1)\)

MOV <Xd|SP>, #<imm>

is equivalent to

ORR <Xd|SP>, XZR, #<imm>

and is the preferred disassembly when \(!\text{MoveWidePreferred}(sf, N, imms, immr)\).

Assembler Symbols

- <Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.
  
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  
  - The values of the data supplied in any of its registers.
  
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:

  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.
MOV (register)

Move (register) copies the value in a source register to the destination register.

This is an alias of ORR (shifted register). This means:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

MOV <Wd>, <Wm>

is equivalent to

ORR <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

MOV <Xd>, <Xm>

is equivalent to

ORR <Xd>, XZR, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MOVK**

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 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>}
```

```python
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<imm>** Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- **<shift>** For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
  - For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

**Operation**

```
bits(datasize) result;
result = X[d];
result<pos+15:pos> = imm16;
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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).

<table>
<thead>
<tr>
<th>bit</th>
<th>opc</th>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
</table>

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; hw != '00')</td>
</tr>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>32-bit</td>
<td>! (IsZero(imm16) &amp; hw != '00') &amp; ! IsOnes(imm16)</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;

result = Zeros();

result<pos+15:pos> = imm16;
result = NOT(result);
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    • The values of the data supplied in any of its registers.
    • The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
</tr>
</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.

```

1 1 1 0 1 0 1 0 0 1 1 | op0 | CRn | CRm | op2 | Rt
```

MRS <Xt>, (<systemreg>|$<op0>_<op1>_<Cn>_<Cm>_<op2>)

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);
```

```
integer t = UInt(Rt);
integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in 'AArch64 System Registers' in the System Register XML.

<op0> Is an unsigned immediate, encoded in "o0":

```
<table>
<thead>
<tr>
<th>&lt;op0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

```

Operation

X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 *FEAT_SBS* is implemented, PSTATE.SBS.
- If *FEAT_PAN* is implemented, PSTATE.PAN.
- If *FEAT_UAO* is implemented, PSTATE.UAO.
- If *FEAT_DIT* is implemented, PSTATE.DIT.
- If *FEAT_MTE* is implemented, PSTATE.TCO.

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | op1 |   | CRm | op2 |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 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 | 0 | 1 | 0 | 0 | CRm | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
```
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');
bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
  when '00x'
    min_EL = EL1;
  when '010'
    min_EL = EL1;
  when '011'
    min_EL = EL0;
  when '100'
    min_EL = EL2;
  when '101' UNDEFINED;
  when '110'
    min_EL = EL3;
  when '111'
    min_EL = EL1;
    need_secure = TRUE;

if UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && !IsSecure()) then
  UNDEFINED;

PSTATEField field;

case op1:op2 of
  when '000 011'
    if !HaveUAOExt() then
      UNDEFINED;
    field = PSTATEField_UAO;
  when '000 100'
    if !HavePANExt() then
      UNDEFINED;
    field = PSTATEField_PAN;
  when '000 101'
    if !HaveDITExt() then
      UNDEFINED;
    field = PSTATEField_DIT;
  when '011 010'
    if !HaveMTEExt() then
      UNDEFINED;
    field = PSTATEField_TCO;
  when '011 011'
    if !HaveSSBSExt() then
      UNDEFINED;
    field = PSTATEField_SSBS;
  otherwise UNDEFINED;

  // Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
  if PSTATE.EL == EL0 && field IN {PSTATEField_DAFIFSet, PSTATEField_DAFIFClr} then
    if !ELUsingAAArch32(EL1) && (EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') || SCTLR_EL1.UMA == '0') then
      if EL2Enabled() && !ELUsingAAArch32(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>FEAT_UAO</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
<td>PAN</td>
<td>FEAT_PAN</td>
</tr>
<tr>
<td>000</td>
<td>101</td>
<td>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>FEAT_SSBS</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>DIT</td>
<td>FEAT_DIT</td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>100</td>
<td>TCO</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>101</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
<td>DAIFSet</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>111</td>
<td>DAIFClr</td>
<td>-</td>
</tr>
<tr>
<td>1xx</td>
<td>xxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
</tbody>
</table>

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

case field of

when PSTATEField_SSBS
    PSTATE.SSBS = CRm<0>;
when PSTATEField_SP
    PSTATE.SP = CRm<0>;
when PSTATEField_DAIFSet
    PSTATE.D = PSTATE.D OR CRm<3>;
    PSTATE.A = PSTATE.A OR CRm<2>;
    PSTATE.I = PSTATE.I OR CRm<1>;
    PSTATE.F = PSTATE.F OR CRm<0>;
when PSTATEField_DAIFClr
    PSTATE.D = PSTATE.D AND NOT(CRm<3>);
    PSTATE.A = PSTATE.A AND NOT(CRm<2>);
    PSTATE.I = PSTATE.I AND NOT(CRm<1>);
    PSTATE.F = PSTATE.F AND NOT(CRm<0>);
when PSTATEField_PAN
    PSTATE.PAN = CRm<0>;
when PSTATEField_UAO
    PSTATE.UAO = CRm<0>;
when PSTATEField_DIT
    PSTATE.DIT = CRm<0>;
when PSTATEField_TCO
    PSTATE.TCO = CRm<0>;
MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>o0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MSR (<systemreg>|S<op0>_<op1>_Cn>_Cm>_op2>, <Xt>)

AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys.crm = UInt(CRm);

Assembler Symbols

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
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>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<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.
<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" 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.crm, sys.crm, sys_op2, X[t]);
**MSUB**

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias **MNEG**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>1</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></td>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MSUB <Wd>, <Wn>, <Wm>, <Wa>

64-bit (sf == 1)

MSUB <Xd>, <Xn>, <Xm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;

**Assembler Symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Wa>`: Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Xa>`: Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNEG</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```
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;

integer result;
result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MUL

Multiply: Rd = Rn * Rm.

This is an alias of MADD. This means:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode for this instruction.

<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>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MUL <Wd>, <Wn>, <Wm>

is equivalent to

MADD <Wd>, <Wn>, <Wm>, WZR

and is always the preferred disassembly.

64-bit (sf == 1)

MUL <Xd>, <Xn>, <Xm>

is equivalent to

MADD <Xd>, <Xn>, <Xm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MADD gives the operational pseudocode for this instruction.
MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

• The encodings in this description are named to match the encodings of ORN (shifted register).
• The description of ORN (shifted register) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 1 0 1 0 1 0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
<th>imm6</th>
<th>1 1 1 1</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)

MVN <Wd>, <Wm>{, <shift> #<amount>}

is equivalent to

ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}

and is always the preferred disassembly.

64-bit (sf == 1)

MVN <Xd>, <Xm>{, <shift> #<amount>}

is equivalent to

ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ○ The values of the data supplied in any of its registers.
• The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of **SUB (shifted register)**. This means:

- The encodings in this description are named to match the encodings of **SUB (shifted register)**.
- The description of **SUB (shifted register)** gives the operational pseudocode for this instruction.

32-bit ($sf == 0$)

NEG $<Wd>$, $<Wm>$\{, $<shift>$ $#$<amount>\}

is equivalent to

SUB $<Wd>$, WZR, $<Wm>$ \{, $<shift>$ $#$<amount>\}

and is always the preferred disassembly.

64-bit ($sf == 1$)

NEG $<Xd>$, $<Xm>$\{, $<shift>$ $#$<amount>\}

is equivalent to

SUB $<Xd>$, XZR, $<Xm>$ \{, $<shift>$ $#$<amount>\}

and is always the preferred disassembly.

**Assembler Symbols**

$<Wd>$ Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

$<Wm>$ Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

$<Xd>$ Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

$<Xm>$ Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

$<shift>$ Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>$&lt;shift&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<amount>$ For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

The description of **SUB (shifted register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

```
<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>!= 11111</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</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>
</tr>
</tbody>
</table>
```

32-bit (sf == 0)

NEGS <Wd>, <Wm>{, <shift> #<amount>}

is equivalent to

SUBS <Wd>, WZR, <Wm>{, <shift> #<amount>}

and is always the preferred disassembly.

64-bit (sf == 1)

NEGS <Xd>, <Xm>{, <shift> #<amount>}

is equivalent to

SUBS <Xd>, XZR, <Xm>{, <shift> #<amount>}

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

NGC <Wd>, <Wm>

is equivalent to

SBC <Wd>, WZR, <Wm>

and is always the preferred disassembly.

### 64-bit (sf == 1)

NGC <Xd>, <Xm>

is equivalent to

SBC <Xd>, XZR, <Xm>

and is always the preferred disassembly.

### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

### Operation

The description of SBC gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of \texttt{SBCS}. This means:

- The encodings in this description are named to match the encodings of \texttt{SBCS}.
- The description of \texttt{SBCS} gives the operational pseudocode for this instruction.

32-bit (sf == 0)

\texttt{NGCS <Wd>, <Wm>}

is equivalent to

\texttt{SBCS <Wd>, WZR, <Wm>}

and is always the preferred disassembly.

64-bit (sf == 1)

\texttt{NGCS <Xd>, <Xm>}

is equivalent to

\texttt{SBCS <Xd>, XZR, <Xm>}

and is always the preferred disassembly.

\textbf{Assembler Symbols}

\begin{itemize}
  \item \texttt{<Wd>} Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
  \item \texttt{<Wm>} Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
  \item \texttt{<Xd>} Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
  \item \texttt{<Xm>} Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
\end{itemize}

\textbf{Operation}

The description of \texttt{SBCS} gives the operational pseudocode for this instruction.

\textbf{Operational information}

If \texttt{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
  \end{itemize}
- The response of this instruction to asynchronous exceptions does not vary based on:
  \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
  \end{itemize}
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.

```
31 30 29 28 27 26 25 24 23 22 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 0 0 1 1 1 1

CRm   op2
```

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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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**.

32-bit (sf == 0)

ORN \(<Wd>, <Wn>, <Wm>\{, <shift> \#<amount>\}"

64-bit (sf == 1)

ORN \(<Xd>, <Xn>, <Xm>\{, <shift> \#<amount>\}"

Assembler Symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

\(<shift>\) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

\(<amount>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = \texttt{ShiftReg}(m, \text{shift\_type}, \text{shift\_amount});

operand2 = \texttt{NOT}(operand2);

result = operand1 OR operand2;
\texttt{X}[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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
text
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

**Assembler Symbols**

- `<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<imm>` For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOV (bitmask immediate)</strong></td>
<td>Rn == '11111' &amp;&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 values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:

- 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).

### Assembly Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Wd&gt;</td>
<td>Is the 32-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Wn&gt;</td>
<td>Is the 32-bit name of the first general-purpose source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Wm&gt;</td>
<td>Is the 32-bit name of the second general-purpose source register, encoded in the &quot;Rm&quot; field.</td>
</tr>
<tr>
<td>&lt;Xd&gt;</td>
<td>Is the 64-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Xn&gt;</td>
<td>Is the 64-bit name of the first general-purpose source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Xm&gt;</td>
<td>Is the 64-bit name of the second general-purpose source register, encoded in the &quot;Rm&quot; field.</td>
</tr>
<tr>
<td>&lt;shift&gt;</td>
<td>Is the optional shift to be applied to the final source, defaulting to LSL and encoded in &quot;shift&quot;:</td>
</tr>
<tr>
<td></td>
<td>&lt;shift&gt; &lt;amount&gt; For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the &quot;imm6&quot; field.</td>
</tr>
<tr>
<td></td>
<td>For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the &quot;imm6&quot; field.</td>
</tr>
</tbody>
</table>

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (register)</td>
<td>shift == '00' &amp;&amp; imm6 == '000000' &amp;&amp; Rn == '11111'</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  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | Z  | 0  | 1  | 0  | Rn | Rd |

PACDA (Z == 0)

PACDA <Xd>, <Xn|SP>

PACDZA (Z == 1 && Rn == 1111)

PACDZA <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
  UNDEFINED;
if Z == '0' then // PACDA
  if n == 31 then source_is_sp = TRUE;
else // PACDZA
  if n != 31 then UNDEFINED;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if source_is_sp then
  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  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 0  | 1  | 1  | Rn |   |   |   |   | Rd |

**PACDB (Z == 0)**

PACDB <Xd>, <Xn|SP>

**PACDZB (Z == 1 && Rn == 1111)**

PACDZB <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UNDEFINED;

**Assembler Symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

**Operation**

if source_is_sp then
    X[d] = AddPACDB(X[d], SP[]);
else
    X[d] = AddPACDB(X[d], X[n]);

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 |
|-------------------------|-----------------|-----------------|-----------------|
| Rm                      | Rn              | Rd              |

PACGA \(<X_d>\), \(<X_n>\), \(<X_m|SP>\)

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

\(<X_d>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<X_n>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(<X_m|SP>\) 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
  \(X[d] = \text{AddPACGA}(X[n], \text{SP})\);
else
  \(X[d] = \text{AddPACGA}(X[n], X[m])\);
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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 0  | 0  | 0  | Rn | Rd |

### PACIA (Z == 0)

PACIA <Xd>, <Xn|SP>

### PACIZA (Z == 1 && Rn == 11111)

PACIZA <Xd>

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | x | 1  | 0  | 0  | x | 1  | 1  | 1  | 1  | 1  | CRm | op2 |
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm|op2 of
    when '0011 000'    // PACIAZ
        d = 30;
        n = 31;
    when '0011 001'    // PACIASP
        d = 30;
        source_is_sp = TRUE;
        if HaveBTIExt() then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIASP instruction.
            SetBTypeCompatible(BTypeCompatible_PACIXSP());
    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
        \[X[d] = \text{AddPACIA}(X[d], SP)\];
    else
        \[X[d] = \text{AddPACIA}(X[d], X[n])\];

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 0  | 0  | 1  | Rn | 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  | x  | 1  | 0  | 1  | x  | 1  | 1  | 1  | 1  | 1  | CRm | op2 |
PACIB1716 (CRm == 0001 && op2 == 010)

PACIB

PACIBSP (CRm == 0011 && op2 == 011)

PACIBSP

PACIBZ (CRm == 0011 && op2 == 010)

PACIBZ

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 010'    // PACIBZ
    d = 30;
    n = 31;
  when '0011 011'    // PACIBSP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt() then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIBSP instruction.
      SetBTypeCompatible(BTypeCompatible_PACIBSP());
  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]);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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`.

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 1 0 0 1 1 0 | imm12 | Rn  | Rt |
| size | opc |
```

`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
    SetTagCheckedInstruction(FALSE);

bits(64) address;
if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;
Prefetch(address, t<4:0>);
Prefetch Memory (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 = \texttt{PC}[i] + \text{offset};

\textbf{if} \texttt{HaveMTEExt()} \textbf{then}
\begin{verbatim}
\texttt{SetTagCheckedInstruction}(\text{FALSE});
\end{verbatim}
\textbf{Prefetch}(\text{address, t<4:0>});
**PRFM (register)**

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#). For information about memory accesses, see [Load/Store addressing modes](#).

```
  31  30  29  28  27  26  25  24  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
```

PRFM (<prfop>|#<imm5>), (<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 3 else 0;

**Assemble 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](#).

- `<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>` is the 32-bit name of the general-purpose index register, encoded in the "Rm" field when option<0> is set to 0.

- `<Xm>` is the 64-bit name of the general-purpose index register, encoded in the "Rm" field when option<0> is set to 1.

- `<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>#3</td>
</tr>
</tbody>
</table>

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);
bits(64) address;
if n == 31 then
    address = SP[];
else
    address = X[n];
address = address + offset;
Prefetch(address, t<4:0>);
Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an `PRFUM` instruction is IMPLEMENTATION DEFINED. For more information, see `Prefetch memory`. For information about memory accesses, see `Load/Store addressing modes`.

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.
- `<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);
```
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

bits(64) address;
if \( n = 31 \) then
    address = \( \text{SP}[n] \);
else
    address = \( X[n] \);

address = address + offset;

Prefetch(address, t<4:0>);
**PSB CSYNC**

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

**System**

(ARMv8.2)

```
31 30 29 28 27 26 25 24 23 22 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 1
```

**Operation**

```
if !HaveStatisticalProfiling() then EndOfInstruction();
```

```
ProfilingSynchronizationBarrier();
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
  CRm opc
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  1 1 0 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 1 0 0 1 1 1 1 1
```

// No additional decoding required

Operation

```
SpeculativeStoreBypassBarrierToPA();
```
RBIT

Reverse Bits reverses the bit order in a register.


32-bit (sf == 0)

RBIT <Wd>, <Wn>

64-bit (sf == 1)

RBIT <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(datasize) operand = X[n];
bits(datasize) result;
for i = 0 to datasize-1
    result<datasize-1-i> = operand<i>;
X[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**RET**

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 1 1 0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Z op A M Rn Rm
```

**Assembler Symbols**

<\textit{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**

```
integer n = \texttt{UInt}(Rn);

\texttt{RET \{<Xn>\}}
```

```
bits(64) target = X[n];

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

\texttt{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.
**RETAA, RETAB**

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for RETAA, and key B is used for RETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to LR.

---

**Integer**

(Armv8.3)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
| 1 1 0 1 0 1 1 0 | 1 0 | 1 1 1 1 1 | 0 0 0 | 0 | 1 | M | 1 1 1 1 1 | 1 1 1 1 1
```

**RETAA (M == 0)**

RETAA

**RETAB (M == 1)**

RETAB

boolean use_key_a = (M == '0');

if !HavePACExt() then
    UNDEFINED;

**Operation**

```
bits(64) target = X[30];
bits(64) modifier = SP[];
if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);
// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';
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 |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opc | Rn | Rd |

32-bit (sf == 0 && opc == 10)

REV <Wd>, <Wn>

64-bit (sf == 1 && opc == 11)

REV <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
  rev_index = rev_index - 8;
X[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**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**

```plaintext
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

\[\begin{array}{ccccccccccccccccccccccc}
\hline
1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \hline
\end{array}\]

\text{Rn} \quad \text{Rd} \\
\text{sf} \quad \text{opc}

**REV32** 
\(<\text{Xd}>\), \(<\text{Xn}>\)

integer \(d = \text{UInt}(\text{Rd});\)
integer \(n = \text{UInt}(\text{Rn});\)

integer \(\text{datasize} = \text{if sf} == \text{'}1\text{'} \text{then 64 else 32};\)

integer \(\text{container}\_\text{size};\)
case \(\text{opc}\) of
\hspace{1em} when \text{'}00\text{'}
\hspace{1em} \text{Unreachable();}\n\hspace{1em} when \text{'}01\text{'}
\hspace{1.5em} \text{container}\_\text{size} = 16;\n\hspace{1em} when \text{'}10\text{'}
\hspace{1.5em} \text{container}\_\text{size} = 32;\n\hspace{1em} when \text{'}11\text{'}
\hspace{1.5em} \text{if sf} == \text{'}0\text{'} \text{then UNDEFINED};\n\hspace{1.5em} \text{container}\_\text{size} = 64;

**Assembler Symbols**

\(<\text{Xd}>\) \hspace{1em} \text{}Is the 64-bit name of the general-purpose destination register, encoded in the \text{"Rd" field.}
\(<\text{Xn}>\) \hspace{1em} \text{}Is the 64-bit name of the general-purpose source register, encoded in the \text{"Rn" field.}

**Operation**

\[\text{bits(datasize)} \text{ operand} = \text{X}[n];\]
\[\text{bits(datasize)} \text{ result};\]

integer \(\text{containers} = \text{datasize DIV container}\_\text{size};\)
integer \(\text{elements_per_container} = \text{container}\_\text{size DIV 8};\)
integer \(\text{index} = 0;\)
integer \(\text{rev}\_\text{index};\)
for \(c = 0 \text{ to } \text{containers-1}\)
\hspace{1em} \(\text{rev}\_\text{index} = \text{index} + ((\text{elements_per_container} - 1) \times 8);\)
\hspace{1em} for \(e = 0 \text{ to } \text{elements_per_container-1}\)
\hspace{2em} \(\text{result}<\text{rev}\_\text{index+7:rev}\_\text{index}> = \text{operand}<\text{index+7:index}>;\)
\hspace{2em} \(\text{index} = \text{index} + 8;\)
\hspace{2em} \(\text{rev}\_\text{index} = \text{rev}\_\text{index} - 8;\)
\[\text{X}[d] = \text{result};\]

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register. When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is \textit{optional} whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This is a pseudo-instruction of \texttt{REV}. This means:

- The encodings in this description are named to match the encodings of \texttt{REV}.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of \texttt{REV} gives the operational pseudocode for this instruction.

\begin{center}
\begin{tabular}{cccccccccccccccccccccccccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & Rn & Rd & sf & opc \\
\end{tabular}
\end{center}

64-bit

\texttt{REV64 <Xd>, <Xn>} is equivalent to

\texttt{REV <Xd>, <Xn>}

Assembler Symbols

\texttt{<Xd>} Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\texttt{<Xn>} Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of \texttt{REV} gives the operational pseudocode for this instruction.

Operational information

If \texttt{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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**

(\texttt{Armv8.4})

\begin{center}
\begin{tabular}{cccccccccccccccccc}
\hline
1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & & imm6 & 0 & 0 & 0 & 0 & 1 & & Rn & 0 & & mask & sf
\end{tabular}
\end{center}

RMIF $<Xn>$, $\#<shift>$, $\#<mask>$

\begin{verbatim}
if !HaveFlagManipulateExt() then UNDEFINED;
integer lsb = UInt(imm6);
integer n = UInt(Rn);
\end{verbatim}

**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**

\begin{verbatim}
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>;
\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.
ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode for this instruction.

| sf | 0 0 1 0 0 1 1 1 | N 0 | Rm | imms | Rn | Rd |

32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

ROR <Wd>, <Ws>, #<shift>

is equivalent to

EXTR <Wd>, <Ws>, <Ws>, #<shift>

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1 && N == 1)

ROR <Xd>, <Xs>, #<shift>

is equivalent to

EXTR <Xd>, <Xs>, <Xs>, #<shift>

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Ws> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xs> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <shift> For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field.
  For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 0 1 0 1 1 0</th>
<th>Rm</th>
<th>0 0 1 0 1 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ROR <Wd>, <Wn>, <Wm>

is equivalent to

RORV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

ROR <Xd>, <Xn>, <Xm>

is equivalent to

RORV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RORV

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).

\[
\begin{array}{|c|c|c|}
\hline
31 & 30 & 29 \\
\hline
28 & 27 & 26 \\
\hline
25 & 24 & 23 \\
\hline
22 & 21 & 20 \\
\hline
19 & 18 & 17 \\
\hline
16 & 15 & 14 \\
\hline
13 & 12 & 11 \\
\hline
10 & 9 & 8 \\
\hline
7 & 6 & 5 \\
\hline
4 & 3 & 2 \\
\hline
1 & 0 \\
\hline
\end{array}
\]

op2

32-bit (sf == 0)

RORV \langle Wd \rangle, \langle Wn \rangle, \langle Wm \rangle

64-bit (sf == 1)

RORV \langle Xd \rangle, \langle Xn \rangle, \langle Xm \rangle

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

\langle Wd \rangle Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
\langle Wn \rangle Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
\langle Wm \rangle 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.
\langle Xd \rangle Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\langle Xn \rangle Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
\langle Xm \rangle 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.

```
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**.

32-bit \((sf == 0)\)

\[
\begin{array}{ccccccccccccccccccccccc}
|sf| & 1 & 1 & 1 & 0 & 0 & 0 & 0 & |Rm| & 0 & 0 & 0 & 0 & 0 & |Rn| & |Rd| & \\
\end{array}
\]

**op S**

64-bit \((sf == 1)\)

\[
\begin{array}{ccccccccccccccccccccccc}
|sf| & 1 & 1 & 1 & 0 & 0 & 0 & 0 & |Rm| & 0 & 0 & 0 & 0 & 0 & |Rn| & |Rd| & \\
\end{array}
\]

**<Wd>, <Wn>, <Wm>**

**<Xd>, <Xn>, <Xm>**

integer \(d = \text{UInt}(Rd)\);

integer \(n = \text{UInt}(Rn)\);

integer \(m = \text{UInt}(Rm)\);

integer \(\text{datasize} = \text{if } sf == '1' \text{ then } 64 \text{ else } 32\);

**Assembler Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGCS</td>
<td>Rn == '1111'</td>
</tr>
</tbody>
</table>

**Operation**

\[
\text{bits(datasize) result;}
\text{bits(datasize) operand1} = X[n];
\text{bits(datasize) operand2} = X[m];
\text{bits(4) nzcv;}
\text{operand2} = \text{NOT(operand2);}\\
\text{(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);}\\
\text{PSTATE.}<N,Z,C,V> = nzcv;\\
X[d] = result;
\]

**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.
SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of \(<width>\) bits from the least significant bits of the source register to bit position \(<lsb>\) of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 & N == 0)

SBFIZ \(<Wd>\), \(<Wn>\), #\(<lsb>\), #\(<width>\)

is equivalent to

SBFM \(<Wd>\), \(<Wn>\), #(-\(<lsb>\) MOD 32), #(#\(<width>-1)\)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 & N == 1)

SBFIZ \(<Xd>\), \(<Xn>\), #\(<lsb>\), #\(<width>\)

is equivalent to

SBFM \(<Xd>\), \(<Xn>\), #(-\(<lsb>\) MOD 64), #(#\(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler Symbols

\(<Wd>\)  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Wn>\)  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
\(<Xd>\)  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Xn>\)  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
\(<lsb>\)  For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
          For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
\(<width>\)  For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-\(<lsb>\).
          For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-\(<lsb>\).

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly. If \(<imms>\) is greater than or equal to \(<immr>\), this copies a bitfield of \((<imms>-<immr>+1)\) bits starting from bit position \(<immr>\) in the source register to the least significant bits of the destination register. If \(<imms>\) is less than \(<immr>\), this copies a bitfield of \((<imms>+1)\) bits from the least significant bits of the source register to bit position \((\text{regsize}-<immr>)\) of the destination register, where \(\text{regsize}\) is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases \(\text{ASR} \ (\text{immediate})\), \(\text{SBFIZ}\), \(\text{SBFX}\), \(\text{SXTB}\), \(\text{SXTH}\), and \(\text{SXTW}\).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rd</th>
<th>Rn</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)

\(\text{SBFM} \ <Wd>, \ <Wn>, \ #<immr>, \ #<imms>\)

### 64-bit (sf == 1 && N == 1)

\(\text{SBFM} \ <Xd>, \ <Xn>, \ #<immr>, \ #<imms>\)

integer \(d = \text{UInt}(\text{Rd});\)
integer \(n = \text{UInt}(\text{Rn});\)
integer \(\text{datasize} = \text{if sf} == '1' \text{then 64 else 32};\)
integer \(R;\)
integer \(S;\)
bits(\(\text{datasize}\)) \(wmask;\)
bits(\(\text{datasize}\)) \(tmask;\)

if \(sf == '1' \&\& N != '1'\) then UNDEFINED;
if \(sf == '0' \&\& (N != '0' || \text{imms}<5> != '0' || \text{immr}<5> != '0' )\) then UNDEFINED;

\(R = \text{UInt}(\text{immr});\)
\(S = \text{UInt}(\text{imms});\)
\((\text{wmask}, \text{tmask}) = \text{DecodeBitMasks}(N, \text{imms}, \text{immr}, \text{FALSE});\)

### 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>(\text{ASR} \ (\text{immediate}))</td>
<td>32-bit</td>
<td>\text{imms} == '011111'</td>
</tr>
<tr>
<td>Alias</td>
<td>Of variant</td>
<td>Is preferred when</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>ASR (immediate)</td>
<td>64-bit</td>
<td>imms == '111111'</td>
</tr>
<tr>
<td>SBFIZ</td>
<td></td>
<td>UInt(imms) &lt; UInt(immr)</td>
</tr>
<tr>
<td>SBFX</td>
<td></td>
<td>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</td>
</tr>
<tr>
<td>SXTB</td>
<td></td>
<td>immr == '000000' &amp;&amp; imms == '000111'</td>
</tr>
<tr>
<td>SXTH</td>
<td></td>
<td>immr == '000000' &amp;&amp; imms == '001111'</td>
</tr>
<tr>
<td>SXTW</td>
<td></td>
<td>immr == '000000' &amp;&amp; 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.
SBFX

Signed Bitfield Extract copies a bitfield of $<width>$ bits starting from bit position $<lsb>$ in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| sf | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | N | |.immr| |.imms| |.Rn| |.Rd|
   
   opc
```

32-bit ($sf == 0 && N == 0$)

SBFX $<Wd>$, $<Wn>$, $#<lsb>$, $#<width>$

is equivalent to

SBFM $<Wd>$, $<Wn>$, $#<lsb>$, #($<lsb>$+$<width>$-1)

and is the preferred disassembly when BFXPreferred($sf$, opc$<1>$, imms, immr).

64-bit ($sf == 1 && N == 1$)

SBFX $<Xd>$, $<Xn>$, $#<lsb>$, $#<width>$

is equivalent to

SBFM $<Xd>$, $<Xn>$, $#<lsb>$, #($<lsb>$+$<width>$-1)

and is the preferred disassembly when BFXPreferred($sf$, opc$<1>$, imms, immr).

**Assembler Symbols**

- $<Wd>$: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<Wn>$: Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- $<Xd>$: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<Xn>$: Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- $<lsb>$: For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- $<width>$: For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-$<lsb>$.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-$<lsb>$.

**Operation**

The description of SBFM gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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.

32-bit (sf == 0)

SDIV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SDIV <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;
if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));
X[d] = result<datasize-1:0>;
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  | 0  | 0  | 0  | 0  | 0  | 0  |
| sf |

SETF8 (sz == 0)

SETF8 <Wn>

SETF16 (sz == 1)

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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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  |

CRm  op2

SEV

// Empty.

**Operation**

```c
SendEvent();
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SEVL

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 \texttt{WFE} instruction.

\[
\begin{array}{c}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

CRm op2

SEVL

// Empty.

Operation

\texttt{SendEventLocal();}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 1 0 1 1 0 0 1 Rm 0 Ra Rn Rd</td>
</tr>
</tbody>
</table>

SMADDL <Xd>, <Wn>, <Wm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

Xd Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
Wn Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
Wm Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
Xa Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMULL</td>
<td>Ra == '1111'</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.
Secure Monitor Call causes an exception to EL3.
SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.
If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC value 0x17, that is taken to EL3.
If the value of HCR_EL2.TSC is 1 and EL2 is enabled in the current Security state, execution of an SMC instruction at EL1 generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD. For more information, see Traps to EL2 of Non-secure EL1 execution of SMC instructions.
If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 0 0 1 1

SMC #<imm>

// Empty.
```

**Assembler Symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

**Operation**

```
AArch64.CheckForSMCUndefOrTrap(imm16);
AArch64.CallSecureMonitor(imm16);
```
**SMNEGL**

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of **SMSUBL**. This means:

- The encodings in this description are named to match the encodings of **SMSUBL**.
- The description of **SMSUBL** gives the operational pseudocode for this instruction.

![Instruction Format](image)

<table>
<thead>
<tr>
<th>SMNEGL &lt;Xd&gt;, &lt;Wn&gt;, &lt;Wm&gt;</th>
</tr>
</thead>
</table>

is equivalent to

**SMSUBL <Xd>, <Wn>, <Wm>, XZR**

and is always the preferred disassembly.

**Assembler Symbols**

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of **SMSUBL** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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 Encoding](image)

**Assembly**

```assembly
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

**Assembler Symbols**

- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Xa>` is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMNEGL</td>
<td>Ra == ’11111’</td>
</tr>
</tbody>
</table>

**Operation**

```assembly
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d] = result<63:0>;
```

**Operational Information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

![Register Diagram]

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

```plaintext
bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;
result = Int(operand1, FALSE) * Int(operand2, FALSE);
X[d] = result<127:64>;
```

Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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.

```
1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 1
```

SSBB

// No additional decoding required

**Operation**

```c
SpeculativeStoreBypassBarrierToVA();
```
ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.
This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

**Post-index**
(Armv8.5)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1 1 0 1 imm9 0 1 Xn Xt</td>
</tr>
</tbody>
</table>

```
ST2G <Xt|SP>, [<Xn|SP>], #<simm>
```

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

**Pre-index**
(Armv8.5)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1 1 0 1 imm9 1 1 Xn Xt</td>
</tr>
</tbody>
</table>

```
ST2G <Xt|SP>, [<Xn|SP>], #<simm>!
```

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

**Signed offset**
(Armv8.5)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1 1 0 1 imm9 1 0 Xn Xt</td>
</tr>
</tbody>
</table>

```
ST2G <Xt|SP>, [<Xn|SP>{, #<simm>}
```

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

**Assembler Symbols**

<Xt|SP> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

```plaintext
bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;
if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ST64B**

Single-copy Atomic 64-byte Store without Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location. The data that is stored is atomic and is required to be 64-byte-aligned.

### Integer

(Armv8.7)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | Rn | | Rt |

ST64B <Xt>, [<Xn|SP> {,#0}]

if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

### Assembler Symbols

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

**CheckLDS64BEnabled();**

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
  value = X[t+i];
  if BigEndian(acctype) then value = BigEndianReverse(value);
  data<63+64*i:64*i> = value;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

MemStore64B(address, data, acctype);
ST64BV

Single-copy Atomic 64-byte Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer
(Armv8.7)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

ST64BV <Xs>, <Xt>, [<Xn|SP>]

if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;

Assembler Symbols

<Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.
The value returned is:

0  If the operation updates memory.

1  If the operation fails to update memory.

0xFFFFFFFF_FFFFFFFF
If the memory location accessed does not support this instruction.
If XZR is used, then the return value is ignored.

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

CheckST64BVEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
    value = X[t+i];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

status = MemStore64BWithRet(address, data, acctype);

if s != 31 then X[s] = status;
Single-copy Atomic 64-byte EL0 Store with Return stores eight 64-bit doublewords from consecutive registers, \(X_t\) to \(X(t+7)\), to a memory location, with the bottom 32 bits taken from \textit{ACCDATA\_EL1}, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

**Integer**

\textit{(Armv8.7)}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rs | 1  | 0  | 1  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 1  | Rs | 1  | 0  | 1  | 0  | 0  | 0  | Rn |

If \(!\text{HaveFeatLS64}()\) then UNDEFINED;
if \(Rt<4:3> == \text{‘11’} \text{ or } Rt<0> == \text{‘1’}\) then UNDEFINED;

integer \(n = \text{UInt}(Rn)\);
integer \(t = \text{UInt}(Rt)\);
integer \(s = \text{UInt}(Rs)\);
boolean \(\text{tag\_checked} = n != 31\);

**Assembler Symbols**

\(<X_s>\) Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.
The value returned is:
0\quad If the operation updates memory.
1\quad If the operation fails to update memory.

\textbf{0xFFFFFFFF\_FFFFFFFE}
If the memory location accessed does not support this instruction.

If \(XZR\) is used, then the return value is ignored.

\(<X_t>\) Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<X_n|S>P>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

`CheckST64BV0Enabled();`

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
  value = X[t+i];
  if BigEndian(acctype) then value = BigEndianReverse(value);
  data<63+64*i:64*i> = value;

data<31:0> = ACCDATA_EL1<31:0>;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

status = MemStore64BWithRet(address, data, acctype);

if s != 31 then X[s] = status;
STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD does not have release semantics.
- STADDL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

### Integer (Armv8.1)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x  1 1 1 0 0 0 0  R 1  Rs  0 0 0 0 0 0  Rn 1 1 1 1 1</td>
</tr>
</tbody>
</table>

#### 32-bit LDADD alias (size == 10 & R == 0)

STADD <Ws>, [<Xn|SP>]

is equivalent to

LDADD <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 32-bit LDADDL alias (size == 10 & R == 1)

STADDL <Ws>, [<Xn|SP>]

is equivalent to

LDADDL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDADD alias (size == 11 & R == 0)

STADD <Xs>, [<Xn|SP>]

is equivalent to

LDADD <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDADDL alias (size == 11 & R == 1)

STADDL <Xs>, [<Xn|SP>]

is equivalent to

LDADDL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
## Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB does not have release semantics.
- STADDLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rs | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| size | A | opc | Rn | Rt |

No memory ordering (R == 0)

STADDB <Ws>, [<Xn|SP>]

is equivalent to

LDADDB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STADDLB <Ws>, [<Xn|SP>]

is equivalent to

LDADDLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH does not have release semantics.
- STADDLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

### Integer (Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1 |

**No memory ordering (R == 0)**

STADDH <Ws>, [<Xn|SP>]

is equivalent to

LDADDH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

STADDLH <Ws>, [<Xn|SP>]

is equivalent to

LDADDLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

---

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR does not have release semantics.
- STCLRL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

### Integer (Armv8.1)

<table>
<thead>
<tr>
<th>size</th>
<th>A</th>
<th>opc</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit LDCLR alias (size == 10 && R == 0)

STCLR <Ws>, [<Xn|SP>]

is equivalent to

LDCLR <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 32-bit LDCLRL alias (size == 10 && R == 1)

STCLRL <Ws>, [<Xn|SP>]

is equivalent to

LDCLRL <Ws>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDCLR alias (size == 11 && R == 0)

STCLR <Xs>, [<Xn|SP>]

is equivalent to

LDCLR <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDCLRL alias (size == 11 && R == 1)

STCLRL <Xs>, [<Xn|SP>]

is equivalent to

LDCLRL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**STCLRB, STCLRLB**

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- **STCLRB** does not have release semantics.
- **STCLRLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of *LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB*. This means:

- The encodings in this description are named to match the encodings of *LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB*.
- The description of *LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB* gives the operational pseudocode for this instruction.

### Integer (Armv8.1)

<table>
<thead>
<tr>
<th>No memory ordering (R == 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STCLRB</strong> &lt;Ws&gt;, [&lt;Xn</td>
</tr>
<tr>
<td>is equivalent to</td>
</tr>
<tr>
<td><strong>LDCLRB</strong> &lt;Ws&gt;, WZR, [&lt;Xn</td>
</tr>
<tr>
<td>and is always the preferred disassembly.</td>
</tr>
</tbody>
</table>

### Release (R == 1)

| **STCLRLB** <Ws>, [<Xn|SP>] |
| is equivalent to              |
| **LDCLRLB** <Ws>, WZR, [<Xn|SP>] |
| and is always the preferred disassembly. |

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of *LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB* gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a
bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH does not have release semantics.
- STCLRLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDCLRHL, LDCLRHA, LDCLRALH, LDCLRLH**. This means:

- The encodings in this description are named to match the encodings of **LDCLRHL, LDCLRHA, LDCLRALH, LDCLRLH**.
- The description of **LDCLRHL, LDCLRHA, LDCLRALH, LDCLRLH** gives the operational pseudocode for this instruction.

### Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 0  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1 |

size   A opc Rs Rn Rt

**No memory ordering (R == 0)**

STCLRH `<Ws>` , `[<Xn|SP>]`

is equivalent to

LDCLRHL `<Ws>` , WZR , `[<Xn|SP>]`

and is always the preferred disassembly.

**Release (R == 1)**

STCLRLH `<Ws>` , `[<Xn|SP>]`

is equivalent to

LDCLRHL `<Ws>` , WZR , `[<Xn|SP>]`

and is always the preferred disassembly.

**Assembler Symbols**

* `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the
contents of the memory location, encoded in the "Rs" field.

* `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of **LDCLRHL, LDCLRHA, LDCLRALH, LDCLRLH** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEOR does not have release semantics.
- STEORL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDEOR, LDEORA, LDEORAL, LDEORL. This means:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

**Integer**

(Armv8.1)

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 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   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| size| A   | opc |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Rn  | 1   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | R   | 1   | Rs  | 0   | 0   | 1   | 0   | 0   | R   | 1   | 1   | 1   | 1   | 1   |
```  

32-bit LDEOR alias (size == 10 & R == 0)

STEOR \(<Ws>\), \([<Xn|SP>]\)

is equivalent to

LDEOR \(<Ws>\), WZR, \([<Xn|SP>]\)

and is always the preferred disassembly.

32-bit LDEORL alias (size == 10 & R == 1)

STEORL \(<Ws>\), \([<Xn|SP>]\)

is equivalent to

LDEORL \(<Ws>\), WZR, \([<Xn|SP>]\)

and is always the preferred disassembly.

64-bit LDEOR alias (size == 11 & R == 0)

STEOR \(<Xs>\), \([<Xn|SP>]\)

is equivalent to

LDEOR \(<Xs>\), XZR, \([<Xn|SP>]\)

and is always the preferred disassembly.

64-bit LDEORL alias (size == 11 & R == 1)

STEORL \(<Xs>\), \([<Xn|SP>]\)

is equivalent to

LDEORL \(<Xs>\), XZR, \([<Xn|SP>]\)

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORB does not have release semantics.
- STEORLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Integer

(Armv8.1)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

size A opc Rn Rt
```

No memory ordering (R == 0)

STEORB <Ws>, [<Xn|SP>]

is equivalent to

LDEORB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STEORLB <Ws>, [<Xn|SP>]

is equivalent to

LDEORLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORH does not have release semantics.
- STEORLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH. This means:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

No memory ordering (R == 0)

STEORH <Ws>, [<Xn|SP>]

is equivalent to

LDEORH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STEORLH <Ws>, [<Xn|SP>]

is equivalent to

LDEORLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: **Post-index**, **Pre-index** and **Signed offset**

### Post-index
(Armv8.5)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

STG `<Xt|SP>, [<Xn|SP>!, #<simm>`)

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

### Assembler Symbols

- `<Xt|SP>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- `<simm>` Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

```plaintext
bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  address = address + offset;

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 \texttt{ID\_AA64PFR1\_EL1\_MTE} \neq 0b0010, this instruction is UNDEFINED.

Integer (Armv8.5)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>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>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textbf{STGM} <Xt>, [<Xn|SP>]

if !
\texttt{HaveMTE2Ext}() then UNDEFINED;

integer t = \texttt{UInt}(Xt);

integer n = \texttt{UInt}(Xn);

\textbf{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.

\textbf{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 * (\texttt{UInt}(GMID\_EL1.BS));

address = \texttt{Align}(address, size);

integer count = size \gg \texttt{LOG2\_TAG\_GRANULE};

integer index = \texttt{UInt}(address \ll \texttt{LOG2\_TAG\_GRANULE}+3:\texttt{LOG2\_TAG\_GRANULE});

for i = 0 to count-1
  bits(4) tag = data<(index*4)+3:index*4>;
  \texttt{AArch64\_MemTag}[address, AccType\_NORMAL] = tag;
  address = address + \texttt{TAG\_GRANULE};
  index = index + 1;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

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  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

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  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

### Assembler Symbols

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
For the post-index and pre-index variant: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field.

For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

**Operation**

```plaintext
bits(64) address;
bits(64) data1;
bits(64) data2;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data1 = X[t];
data2 = X[t2];

if !postindex then
    address = address + offset;

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

AArch64.MemTag[address, AccType_NORMAL] = AArch64.AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```
Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

### No offset

(Armv8.1)

```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 | x | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | (1) | (1) | (1) | (1) | 0 | (1) | (1) | (1) | (1) |
| size | L | Rs | o0 | Rt2 |
```

#### 32-bit (size == 10)

STLLR <Wt>, [<Xn|SP>{,#0}]

#### 64-bit (size == 11)

STLLR <Xt>, [<Xn|SP>{,#0}]

```plaintext
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

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = 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.
Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

### No offset

(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**STLLRB**

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
  **SetTagCheckedInstruction**(tag_checked);
if **n** == 31 then
  **CheckSPAlignment**();
  address = **SP**[];
else
  address = **X**[**n**];
data = **X**[**t**];
**Mem**(address, 1, **AccType_LIMITEDORDERED**) = data;
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

No offset
(Armv8.1)

No offset
(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>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>(1</td>
<td>1)</td>
<td>(1</td>
<td>1)</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td></td>
<td></td>
<td>L</td>
<td></td>
<td></td>
<td>Rs</td>
<td>o0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STLLRH <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
data = X[t];
Mem[address, 2, AccType_LIMITEDORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The
instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about
memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0 |
| size | L | Rs | o0 | Rt |

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
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, dbytes, AccType.ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>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>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>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size | L   | Rs  | o0 | Rt2

STLRB <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, 1, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 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  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

STLRH <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bts(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, 2, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.
For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset
(Armv8.4)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | imm9 | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

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

```
ingen = UInt(Rn);
t = UInt(Rt);
datasize = 8 << scale;
tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
SetTagCheckedInstruction(tag_checked);

address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = X[t];
Mem[address, datasize DIV 8, AccType_ORDERED] = data;
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(Armv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 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

(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.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = `UInt`(Rn);
integer t = `UInt`(Rt);

boolean tag_checked = n != 31;

Operation

if `HaveMTEExt()` then
  `SetTagCheckedInstruction`(tag_checked);
bits(64) address;
bits(8) data;
if n == 31 then
  `CheckSPAlignment`();
  address = `SP`[];
else
  address = `X`[n];
address = address + offset;
data = `X`[t];
`Mem`[address, 1, `AccType.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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | imm9 | 0  | 0  | Rn  | 0  | 0  | Rt  | bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = X[t];
Mem[address, 2, AccType.ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

32-bit (sz == 0)

STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);  // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 32;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
        when Constraint_NONE rt unknown = FALSE;    // store original value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == n & n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
            when Constraint_NONE rn unknown = FALSE;    // address is original base
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXP.

Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0  If the operation updates memory.

1  If the operation fails to update memory.

<Xt1>  Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(AccType_ORDEREDATOMIC) 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.
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |

32-bit (size == 10)

STLXR <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STLXR <Ws>, <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_NONE rt_unknown = FALSE; // store original value
        when Constraint_UNDEF Undefined;
        when Constraint_NOP EndOfInstruction();
if s == t & n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NONE rn_unknown = FALSE; // address is original base
        when Constraint_UNDEF Undefined;
        when Constraint_NOP EndOfInstruction();
For information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 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;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

 integer \( n = \text{UInt}(Rn) \);
 integer \( t = \text{UInt}(Rt) \);
 integer \( s = \text{UInt}(Rs) \); // ignored by all loads and store-release

 boolean tag_checked = \( n \neq 31 \);
 boolean rt_unknown = FALSE;
 boolean rn_unknown = FALSE;

 if \( s == t \) then
    Constraint \( c = \text{ConstrainUnpredictable}(\text{Unpredictable\_DATAOVERLAP}) \);
    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 \text{UNDEFINED};
        when Constraint\_NOP \text{EndOfInstruction}();

 if \( s == n \) \&\& \( n \neq 31 \) then
    Constraint \( c = \text{ConstrainUnpredictable}(\text{Unpredictable\_BASEOVERLAP}) \);
    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 \text{UNDEFINED};
        when Constraint\_NOP \text{EndOfInstruction}();

For information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRB.

Assembler Symbols

\(<Ws>\) Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts
If a synchronous Data Abort exception is generated by the execution of this instruction:

• Memory is not updated.
• \(<Ws>\) is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation

bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elif 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 | L | 00 | Rs | 1 | (1) | (1) | (1) | (1) | Rn | Rt |

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;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
  Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
    when Constraint_NONE rt unknown = FALSE;  // store original value
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
if s == t & n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
    when Constraint_NONE rn unknown = FALSE;  // address is original base
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly STLXRH.

Assembler Symbols

\(<Ws>\) Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<Ws>\) is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```plaintext
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see Load/Store addressing modes. For information about Non-temporal pair instructions, see Load/Store Non-temporal pair.

<table>
<thead>
<tr>
<th>31</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>x</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>imm7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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 == 00)

STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]

64-bit (opc == 10)

STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]

// Empty.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bite(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = X[t];
data2 = X[t2];
Mem[address, dbytes, AccType_STREAM] = data1;
Mem[address+dbytes, dbytes, AccType_STREAM] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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**

<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>x</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>imm7</td>
<td></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>
</tr>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<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)

STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>

64-bit (opc == 10)

STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;

**Pre-index**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | imm7|    | Rt2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 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  | 0  | 0  | 1  | imm7|    | Rt2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 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

- `<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as `<imm>/4`. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as `<imm>/4`. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as `<imm>/8`. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as `<imm>/8`.

### Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bite(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

*STP*
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2];
Mem[address, dbytes, AccType_NORMAL] = data1;
Mem[address+dbytes, dbytes, AccType_NORMAL] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**32-bit (size == 10)**

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

64-bit (size == 11)

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**32-bit (size == 10)**

```java
boolean wback = FALSE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

64-bit (size == 11)

```java
boolean wback = FALSE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**32-bit (size == 10)**

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm12, 64);
```

64-bit (size == 11)

```java
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm12, 64);
```
32-bit (size == 10)

STR <Wt>, [<Xn|SP>{, #<pimm>}]  

64-bit (size == 11)

STR <Xt>, [<Xn|SP>{, #<pimm>}]  

boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.  
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.  
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.  
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.  
<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.  
For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer datasize = 8 << scale;  
boolean tag_checked = wback || n != 31;  
boolean rt_unknown = FALSE;  
if wback && n == t && n != 31 then  
c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);  
assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};  
case c of  
when Constraint_NONE  
    rt_unknown = FALSE;  // value stored is original value  
when Constraint_UNKNOWN  
    rt_unknown = TRUE;  // value stored is UNKNOWN  
when Constraint_UNDEF  
    UNDEFINED;  
when Constraint_NOP  
    EndOfInstruction();
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
b�新(data) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STR (register)**

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

```
<table>
<thead>
<tr>
<th>x</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>S</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
</tr>
</tbody>
</table>
```

32-bit (size == 10)

\[
\text{STR } \langle Wt \rangle, \langle Xn|SP \rangle, \langle Wm|Xm \rangle\{, \langle extend \rangle \{\langle amount \rangle\}\}
\]

64-bit (size == 11)

\[
\text{STR } \langle Xt \rangle, \langle Xn|SP \rangle, \langle Wm|Xm \rangle\{, \langle extend \rangle \{\langle amount \rangle\}\}
\]

```plaintext
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**

- \( \langle Wt \rangle \): Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- \( \langle Xt \rangle \): Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- \( \langle Xn|SP \rangle \): Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- \( \langle Wm \rangle \): When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- \( \langle Xm \rangle \): When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- \( \langle extend \rangle \): Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when \( \langle amount \rangle \) is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>( \langle extend \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTW</td>
</tr>
<tr>
<td>001</td>
<td>LSL</td>
</tr>
<tr>
<td>010</td>
<td>SXTW</td>
</tr>
<tr>
<td>011</td>
<td>SXTX</td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

- \( \langle amount \rangle \): For the 32-bit variant: is the index shift amount, optional only when \( \langle extend \rangle \) 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>( \langle amount \rangle )</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 \( \langle extend \rangle \) 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>( \langle amount \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: *Post-index*, *Pre-index* and *Unsigned offset*.

**Post-index**

```
0 0 1 1 1 0 0 0 0 0 0 | imm9 | 0 1 | Rn | Rt
```

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

**Pre-index**

```
0 0 1 1 1 0 0 0 0 0 0 | imm9 | 1 1 | Rn | Rt
```

```java
boolean wback = TRUE;
boolean postindex = FALSE;
b bits(64) offset = SignExtend(imm9, 64);
```

**Unsigned offset**

```
0 0 1 1 1 0 0 1 0 0 0 | imm12 | Rn | Rt
```

```java
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STRB (immediate)*.

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>` Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
integer \( n = \text{UInt}(Rn); \)
integer \( t = \text{UInt}(Rt); \)

boolean tag_checked = wback || \( n \neq 31; \)

boolean rt_unknown = FALSE;

if wback && \( n = t \) && \( n \neq 31 \) then
\[
c = \text{ConstrainUnpredictable(Unpredictable_WBOVERLAPST)};
\]
assert \( c \in \{\text{Constraint\_NONE}, \text{Constraint\_UNKNOWN}, \text{Constraint\_UNDEF}, \text{Constraint\_NOP}\}; \)
case \( c \) of
  when \( \text{Constraint\_NONE} \) \_ rt\_unknown = FALSE;  // value stored is original value
  when \( \text{Constraint\_UNKNOWN} \) \_ rt\_unknown = TRUE;  // value stored is UNKNOWN
  when \( \text{Constraint\_UNDEF} \) \_ UNDEFINED;
  when \( \text{Constraint\_NOP} \) \_ EndOfInstruction();

Operation

if \( \text{HaveMTEExt()} \) then
  \( \text{SetTagCheckedInstruction}(\text{tag\_checked}); \)

bits(64) address;
bits(8) data;

if \( n = 31 \) then
  \( \text{CheckSPAlignment}(); \)
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, \text{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.
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.

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  | 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
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data = X[t];
Mem[address, 1, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|---------------------|---------------------|
| 0 1 1 1 1 0 0 0 0 0 | imm9               | 0 1                 | Rn                  |
| size                | opc                 |                      |
```

STRH <Wt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|---------------------|---------------------|
| 0 1 1 1 1 0 0 0 0 0 | imm9               | 1 1                 | Rn                  |
| size                | opc                 |                      |
```

STRH <Wt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|---------------------|---------------------|
| 0 1 1 1 1 0 0 1 0 0 | imm12              |                      | Rn                  |
| size                | opc                 |                      |
```

STRH <Wt>, [<Xn|SP>{, #<pimm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 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 <pimm>/2.
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];
    Mem[address, 2, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | Rm | option | S  | 1  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| size | opc |

STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

if option<1> == '0' then UNDEFINED;  // sub-word index

ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

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.
STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSET does not have release semantics.
- STSETL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSET, LDSETA, LDSETAL, LDSETL. This means:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

**Integer (Armv8.1)**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 1 | x  | 1 | 1 | 1 | 0 | 0 | 0 | 0 | R | 1 | Rs | 0 | 0 | 1 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | 1 | |
| A   | opc|     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**32-bit LDSET alias (size == 10 && R == 0)**

STSET <Ws>, [<Xn|SP>]

is equivalent to

LDSET <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**32-bit LDSETL alias (size == 10 && R == 1)**

STSETL <Ws>, [<Xn|SP>]

is equivalent to

LDSETL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDSET alias (size == 11 && R == 0)**

STSET <Xs>, [<Xn|SP>]

is equivalent to

LDSET <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDSETL alias (size == 11 && R == 1)**

STSETL <Xs>, [<Xn|SP>]

is equivalent to

LDSETL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational Information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB does not have release semantics.
- STSETLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB. This means:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Integer (Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | R | 1  | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 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  | 1  | 1  | 1  | 1  | 1  | 1  |

size  A  opc  Rn  Rt

No memory ordering (R == 0)

STSETB <Ws>, [<Xn|SP>]

is equivalent to

LDSETB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSETLB <Ws>, [<Xn|SP>]

is equivalent to

LDSETLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH does not have release semantics.
- STSETLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

**Integer**

(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | R | 1 | Rs | 0 | 0 | 1 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | 1 |

size A opc Rn Rt

**No memory ordering (R == 0)**

STSETH <Ws>, [<Xn|SP>]

is equivalent to

LDSETH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

STSETLH <Ws>, [<Xn|SP>]

is equivalent to

LDSETLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX does not have release semantics.
- STSMAXL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL. This means:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|------------------|--------|-------------------|--------|-------------------|--------|-------------------|--------|
| size | A | opc | Rs | 0 1 0 0 0 | Rn | 1 1 1 1 1 |

32-bit LDSMAX alias (size == 10 && R == 0)

STSMAX \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDSMAX \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

32-bit LDSMAXL alias (size == 10 && R == 1)

STSMAXL \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDSMAXL \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

64-bit LDSMAX alias (size == 11 && R == 0)

STSMAX \(<Xs>, [<Xn|SP>]\)

is equivalent to

LDSMAX \(<Xs>, XZR, [<Xn|SP>]\)

and is always the preferred disassembly.

64-bit LDSMAXL alias (size == 11 && R == 1)

STSMAXL \(<Xs>, [<Xn|SP>]\)

is equivalent to

LDSMAXL \(<Xs>, XZR, [<Xn|SP>]\)

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STS MAX B, STS MAX LB**

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- **STSMAXB** does not have release semantics.
- **STSMAXLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB**. This means:

- The encodings in this description are named to match the encodings of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB**.
- The description of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB** gives the operational pseudocode for this instruction.

### Integer (Arm v8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rs | 0  | 1  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  | R  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| size | A | opc | Rt |

**No memory ordering (R == 0)**

**STSMAXB** <Ws>, [Xn|SP]  
is equivalent to  
**LDSMAXB** <Ws>, WZR, [Xn|SP]  
and is always the preferred disassembly.

**Release (R == 1)**

**STSMAXLB** <Ws>, [Xn|SP]  
is equivalent to  
**LDSMAXLB** <Ws>, WZR, [Xn|SP]  
and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STSMAXH, STSMAXLH**

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- **STSMAXH** does not have release semantics.
- **STSMAXLH** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH**. This means:

- The encodings in this description are named to match the encodings of **LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH**.
- The description of **LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH** gives the operational pseudocode for this instruction.

### Integer

*(Armv8.1)*

<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>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>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>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**No memory ordering (R == 0)**

STSMAXH `<Ws>`, [ `<Xn|SP>` ]

is equivalent to

LDSMAXH `<Ws>`, WZR, [ `<Xn|SP>` ]

and is always the preferred disassembly.

**Release (R == 1)**

STSMAXLH `<Ws>`, [ `<Xn|SP>` ]

is equivalent to

LDSMAXLH `<Ws>`, WZR, [ `<Xn|SP>` ]

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STSMIN, STSMINL**

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- **STSMIN** does not have release semantics.
- **STSMINL** stores to memory with release semantics, as described in [*Load-Acquire, Store-Release*](#).

For information about memory accesses see [*Load/Store addressing modes*](#).

This is an alias of **LDSMIN, LDSMINA, LDSMINAL, LDSMINL**. This means:

- The encodings in this description are named to match the encodings of **LDSMIN, LDSMINA, LDSMINAL, LDSMINL**.
- The description of **LDSMIN, LDSMINA, LDSMINAL, LDSMINL** gives the operational pseudocode for this instruction.

### Integer (Armv8.1)

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     | x   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | R   | 1   | Rs  | 0   | 1   | 0   | 1   | 0   | Rn  | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   |

<table>
<thead>
<tr>
<th>size</th>
<th>A</th>
<th>opc</th>
<th>Rt</th>
</tr>
</thead>
</table>

32-bit LDSMIN alias (size == 10 && R == 0)

**STSMIN** <Ws>, [Xn|SP]  
is equivalent to  
**LDSMIN** <Ws>, WZR, [Xn|SP]  
and is always the preferred disassembly.

32-bit LDSMINL alias (size == 10 && R == 1)

**STSMINL** <Ws>, [Xn|SP]  
is equivalent to  
**LDSMINL** <Ws>, WZR, [Xn|SP]  
and is always the preferred disassembly.

64-bit LDSMIN alias (size == 11 && R == 0)

**STSMIN** <Xs>, [Xn|SP]  
is equivalent to  
**LDSMIN** <Xs>, XZR, [Xn|SP]  
and is always the preferred disassembly.

64-bit LDSMINL alias (size == 11 && R == 1)

**STSMINL** <Xs>, [Xn|SP]  
is equivalent to  
**LDSMINL** <Xs>, XZR, [Xn|SP]  
and is always the preferred disassembly.
Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs>  Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINB does not have release semantics.
- STSMINLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB. This means:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

|   |   |   |   |   |   |   | R |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 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 | R | 1 | Rs| 0 | 1 | 0 | 1 | 0 | 0 | Rn| 1 | 1 | 1 | 1 | 1 |   |   |   |   |

size A opc Rt

No memory ordering (R == 0)

STSMINB <Ws>, [<Xn|SP>]

is equivalent to

LDSMINB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSMINLB <Ws>, [<Xn|SP>]

is equivalent to

LDSMINLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSminH, STSminLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSminH does not have release semantics.
- STSminLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSminH, LDSminAH, LDSminALH, LDSminLH. This means:

- The encodings in this description are named to match the encodings of LDSminH, LDSminAH, LDSminALH, LDSminLH.
- The description of LDSminH, LDSminAH, LDSminALH, LDSminLH gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 0 0</td>
</tr>
</tbody>
</table>

size A opc Rn Rt

No memory ordering (R == 0)

STSminH <Ws>, [<Xn|SP>]
is equivalent to

LDSminH <Ws>, WZR, [<Xn|SP>]
and is always the preferred disassembly.

Release (R == 1)

STSminLH <Ws>, [<Xn|SP>]
is equivalent to

LDSminLH <Ws>, WZR, [<Xn|SP>]
and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSminH, LDSminAH, LDSminALH, LDSminLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.E2H, TGE} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

```
31 30 29 28 27 26 25 24 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

1 x 1 1 1 0 0 0 0 0 0 imm9 1 0 Rn Rt
```

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> == '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 datasize = 8 << scale;
boolean tag_checked = n != 31;
```
Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
address = address + offset;
data = X[t];
Mem[address, datasize DIV 8, acctype] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.E2H, TGE is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Asmblr 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
tag_checked = n != 31;
```

**Operation**

```plaintext
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data = X[t];
```

```
Mem[address, 1, accotype] = 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*.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | 0  | 1  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 1 |

** 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
tag_checked = n != 31;
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;
```

**Operation**

```plaintext
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data = X[t];
Mem[address, 2, acctype] = data;
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAX does not have release semantics.
- STUMAXL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL. This means:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

|     | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| A   | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opc |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rt  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit LDUMAX alias (size == 10 && R == 0)

STUMAX <Ws>, [<Xn|SP>]

is equivalent to

LDUMAX <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

32-bit LDUMAXL alias (size == 10 && R == 1)

STUMAXL <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDUMAX alias (size == 11 && R == 0)

STUMAX <Xs>, [<Xn|SP>]

is equivalent to

LDUMAX <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDUMAXL alias (size == 11 && R == 1)

STUMAXL <Xs>, [<Xn|SP>]

is equivalent to

LDUMAXL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXB does not have release semantics.
- STUMAXLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB. This means:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

**Integer**

(ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size A opc Rt

No memory ordering (R == 0)

STUMAXB <Ws>, [<Xn|SP>] is equivalent to

LDUMAXB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STUMAXLB <Ws>, [<Xn|SP>] is equivalent to

LDUMAXLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer.

**Operation**

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH does not have release semantics.
- STUMAXLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

<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>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size    A       opc   Rt

No memory ordering (R == 0)

STUMAXH <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STUMAXLH <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STUMIN, STUMINL**

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMIN does not have release semantics.
- STUMINL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of *LDUMIN, LDUMINA, LDUMINAL, LDUMINL*. This means:

- The encodings in this description are named to match the encodings of *LDUMIN, LDUMINA, LDUMINAL, LDUMINL*.
- The description of *LDUMIN, LDUMINA, LDUMINAL, LDUMINL* gives the operational pseudocode for this instruction.

---

**Integer**

(Armv8.1)

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 x | 1   | 1   | 1   | 0   | 0   | 0   | 0   | R   | 1   | Rs  | 0   | 1   | 1   | 1   | 0   | 0   | Rn  | 1   | 1   | 1   | 1   | 1   | R   |

**32-bit LDUMIN alias (size == 10 && R == 0)**

STUMIN $<Ws>$, $<[Xn]\mid SP>$

is equivalent to

LDUMIN $<Ws>$, WZR, $<[Xn]\mid SP>$

and is always the preferred disassembly.

**32-bit LDUMINL alias (size == 10 && R == 1)**

STUMINL $<Ws>$, $<[Xn]\mid SP>$

is equivalent to

LDUMINL $<Ws>$, WZR, $<[Xn]\mid SP>$

and is always the preferred disassembly.

**64-bit LDUMIN alias (size == 11 && R == 0)**

STUMIN $<Xs>$, $<[Xn]\mid SP>$

is equivalent to

LDUMIN $<Xs>$, XZR, $<[Xn]\mid SP>$

and is always the preferred disassembly.

**64-bit LDUMINL alias (size == 11 && R == 1)**

STUMINL $<Xs>$, $<[Xn]\mid SP>$

is equivalent to

LDUMINL $<Xs>$, XZR, $<[Xn]\mid SP>$

and is always the preferred disassembly.
Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs>  Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINI, gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- **STUMINB** does not have release semantics.
- **STUMINLB** stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB. This means:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

### Integer
(Armv8.1)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|-----------------|-----------------|
| 0 0 1 1 1 0 0 0 0 R 1 | Rs              | 0 1 1 1 0 0    |
| size                   | A               | opc             |
| Rn                     | 1 1 1 1 1 1    |
```

**No memory ordering (R == 0)**

STUMINB <Ws>, [<Xn|SP>]

is equivalent to

LDUMINB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

STUMINLB <Ws>, [<Xn|SP>]

is equivalent to

LDUMINLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINH does not have release semantics.
- STUMINLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R | 1 | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

size A opc Rn Rt

No memory ordering (R == 0)

STUMINH <Ws>, [<Xn|SP>]

is equivalent to

LDUMINH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STUMINLH <Ws>, [<Xn|SP>]

is equivalent to

LDUMINLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | imm9 | 0 | 0 | Rn | Rt |

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
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;
if n == 31 then
    CheckSAPAlignment();
    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.

\[
\begin{array}{ccccccccccccccccccc}
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & imm9 & 0 & 0 & Rn & 0 & 0 & Rt \\
\end{array}
\]

size     opc

STURB \(<Wt>\), \(<Xn|SP>\{, \#<\text{simm}>\} \)

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.
\(<\text{simm}>\)  Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

\[
\begin{align*}
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } t &= \text{UInt}(Rt); \\
\text{boolean } tag\_checked &= n \neq 31;
\end{align*}
\]

Operation

\[
\begin{align*}
\text{if } \text{HaveMTEExt}() \text{ then} \\
&\quad \text{SetTagCheckedInstruction}(tag\_checked); \\
\text{bits(64) address;} \\
\text{bits(8) data;}
\end{align*}
\]

\[
\begin{align*}
\text{if } n &= 31 \text{ then} \\
&\quad \text{CheckSPAlignment}(); \\
&\quad \text{address} = SP[]; \\
\text{else} \\
&\quad \text{address} = X[n]; \\
\text{address} = \text{address} + \text{offset};
\end{align*}
\]

\[
\begin{align*}
\text{data} &= X[t]; \\
\text{Mem}[\text{address}, 1, \text{AccType\_NORMAL}] &= \text{data};
\end{align*}
\]

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|---------------|---------------|--------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 1 1 1 1 0 0 0 0 0 0 0 | imm9 | 0 0 | Rn | Rt |

size        opc

STURH \(<W_t>, [<X_n|SP>\{, #<\text{simm}>\}]

bits(64) offset = \text{SignExtend}(\text{imm9}, 64);

**Assembler Symbols**

\(<W_t>\)  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<X_n|SP>\)  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{simm}>\)  Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

\[
\begin{align*}
\text{integer } n & = \text{UInt}(\text{Rn}); \\
\text{integer } t & = \text{UInt}(\text{Rt}); \\
\text{boolean tag\_checked} & = n \neq 31;
\end{align*}
\]

**Operation**

\[
\begin{align*}
\text{if HaveMTEExt()} & \text{ then} \\
& \quad \text{SetTagCheckedInstruction}(\text{tag\_checked}); \\
\text{bits(64) address;} \\
\text{bits(16) data;} \\
\text{if } n == 31 & \text{ then} \\
& \quad \text{CheckSPAlignment();} \\
& \quad \text{address} = \text{SP}[]; \\
\text{else} \\
& \quad \text{address} = X[n]; \\
\text{address} & = \text{address} + \text{offset}; \\
\text{data} & = X[t]; \\
\text{Mem}[\text{address}, 2, \text{AccType\_NORMAL}] & = \text{data};
\end{align*}
\]

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses see Load/Store addressing modes.

32-bit (sz == 0)

STXP `<Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]`

64-bit (sz == 1)

STXP `<Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]`

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
  Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
    when Constraint_NONE rt_unknown = FALSE; // store original value
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  endcase;
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();
  endcase;

For information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXP.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:

• Memory is not updated.
• <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

• If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
• Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
endif

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(AccType_ATOMIC) then el1:el2 else el2:el1;
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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| L  | Rs | 0  | (1)| (1)| (1)| (1)| (1)| (1)| (1)| (0)|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

32-bit (size == 10)

STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);  // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
  Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
    when Constraint_NONE rt unknown = FALSE;    // store original value
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  end;
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;

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.
- \(<W_s>\) 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 \(\text{AArch64.ExclusiveMonitorsPass}()\) returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If \(\text{AArch64.ExclusiveMonitorsPass}()\) returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if \(\text{HaveMTEExt}()\) then
    \text{SetTagCheckedInstruction}(tag_checked);

if n == 31 then
    \text{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 \(\text{AArch64.ExclusiveMonitorsPass}(\text{address}, \text{dbytes})\) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    \text{Mem}[\text{address}, \text{dbytes}, \text{AccType_ATOMIC}] = data;
    status = \text{ExclusiveMonitorsStatus}();
    X[s] = \text{ZeroExtend}(\text{status}, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. For information about memory accesses see *Load/Store addressing modes*.

```
<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**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;
- boolean rt_unknown = FALSE;
- boolean rn_unknown = FALSE;

  if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
    when Constraint_NONE rt_unknown = FALSE; // store original value
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
    when Constraint_NONE rn_unknown = FALSE; // address is original base
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *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;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];

bit status = '1';

// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_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.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

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;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

**Post-index**
*(Armv8.5)*

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

STZ2G <Xt|SP>, [<Xn|SP>{, #<simm>}

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
built(64) offset = LSL(SignExtend(im9, 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

STZ2G <Xt|SP>, [<Xn|SP>{, #<simm>}

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
built(64) offset = LSL(SignExtend(im9, 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

STZ2G <Xt|SP>, [<Xn|SP>{, #<simm>}

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
built(64) offset = LSL(SignExtend(im9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn> 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 "im9" field.
Operation

bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
**STZG**

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

**Post-index (Armv8.5)**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 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>imm9</th>
<th>0</th>
<th>1</th>
<th>Xn</th>
<th>Xt</th>
</tr>
</thead>
</table>

STZG `<Xt|SP>`, `<Xn|SP>`, #<simm>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

**Pre-index (Armv8.5)**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 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>imm9</th>
<th>1</th>
<th>1</th>
<th>Xn</th>
<th>Xt</th>
</tr>
</thead>
</table>

STZG `<Xt|SP>`, `<Xn|SP>`, #<simm>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

**Signed offset (Armv8.5)**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 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>imm9</th>
<th>1</th>
<th>0</th>
<th>Xn</th>
<th>Xt</th>
</tr>
</thead>
</table>

STZG `<Xt|SP>`, `<Xn|SP>{, #<simm>}

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

**Assembler Symbols**

<Xt|SP> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
**STZGM**

Store Tag and Zero Multiple writes a naturally aligned block of \( N \) Allocation Tags and stores zero to the associated data locations, where the size of \( N \) is identified in DCZID_EL0.BS, and the Allocation Tag written to address \( A \) is taken from the source register bits<3:0>.

This instruction is **UNDEFINED** at EL0.

This instruction generates an Unchecked access.

If **ID_AA64FPRF1_EL1.MTE** \( \neq \) 0b0010, this instruction is **UNDEFINED**.

**Integer**

(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Xn  | Xt  |

\[ \text{STZGM } <Xt>, [<Xn|SP>] \]

if \(!\text{HaveMTE2Ext()}\) then **UNDEFINED**;

integer \( t = \text{UInt}(Xt); \)

integer \( n = \text{UInt}(Xn); \)

**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 \( \text{PSTATE.EL} == \text{EL0} \) then **UNDEFINED**;

\[ \text{bits(64) data } = X[t]; \]

\[ \text{bits(4) tag } = \text{data}<3:0>; \]

\[ \text{bits(64) address}; \]

if \( n == 31 \) then

\[ \text{CheckSPAlignment();} \]

\[ \text{address } = \text{SP}[]; \]

else

\[ \text{address } = X[n]; \]

integer \( \text{size } = 4 * (2 ^ \text{(UInt(DCZID_EL0.BS))}); \)

\[ \text{address } = \text{Align(address, size)}; \]

integer \( \text{count } = \text{size } \gg \text{LOG2_TAG_GRANULE}; \)

for \( i = 0 \) to \( \text{count-1} \)

\[ \text{AArch64.MemTag[address, AccType_NORMAL] } = \text{tag}; \]

\[ \text{Mem[address, TAG_GRANULE, AccType_NORMAL] } = \text{Zeros}(8 * \text{TAG_GRANULE}); \]

\[ \text{address } = \text{address } + \text{TAG_GRANULE}; \]

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
### SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword.

#### 32-bit (sf == 0)

SUB `<Wd|WSP>`, `<Wn|WSP>`, `<Wm>`{, `<extend>` {#<amount>}}

#### 64-bit (sf == 1)

SUB `<Xd|SP>`, `<Xn|SP>`, `<R><m>`{, `<extend>` {#<amount>}}

Integer `d` = UInt(Rd);
Integer `n` = UInt(Rn);
Integer `m` = UInt(Rm);
Integer `datasize` = if `sf` == '1' then 64 else 32;

```python
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'.

---

**Page 512**
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.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>sh</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

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>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>5</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)

SUB `<Wd>, <Wn>, <Wm>{, <shift> #<amount>}`

64-bit (sf == 1)

SUB `<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>NEG (shifted register)</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>
Operation

\begin{verbatim}
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;
\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.
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  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op3 | 1 | 1 | 0 | 1 | 0 | 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**

bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
  rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
  rtag = '0000';
(result, -) = AddWithCarry(operand1, NOT(offset), '1');
result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, 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  | Xm | 0  | 0  | 0  | 0  | 0  | Xn | Xd |

SUBP <Xd>, <Xn|SP>, <Xm|SP>

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

**Assembler Symbols**

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- **<Xn|SP>** Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- **<Xm|SP>** Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

**Operation**

```plaintext
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');
X[d] = result;
```
**SUBPS**

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias **CMPP**.

**Integer**

*(Armv8.5)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**Asm**

**SUBPS** `<Xd>`, `<Xn|SP>`, `<Xm|SP>`

integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

**Assembler Symbols**

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<Xm|SP>` Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPP</td>
<td>S == '1' &amp; Xd == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

- `bits(64) operand1 = if n == 31 then SP[] else X[n];`
- `bits(64) operand2 = if m == 31 then SP[] else X[m];`
- `operand1 = SignExtend(operand1<55:8>, 64);`
- `operand2 = SignExtend(operand2<55:8>, 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;`
**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>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>option</td>
<td>imm3</td>
<td>Rn</td>
<td>Rd</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>}}
```

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

**Assembler Symbols**

- **<Wd>** 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>UTXC</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>CMP (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SUBS (immediate)**

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **CMP (immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>sh</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>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|WSP>, #<imm>{, <shift>}

**64-bit (sf == 1)**

SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}

```plaintext
generate
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);
generate```

**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
generate
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;
generate```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUBS (shifted register)**

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases **CMP (shifted register)**, and **NEGS**.

32-bit (sf == 0)

SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

<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.

**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":

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP (shifted register)</strong></td>
<td>Rd == '11111'</td>
</tr>
<tr>
<td><strong>NEGS</strong></td>
<td>Rn == '11111' &amp; Rd != '11111'</td>
</tr>
</tbody>
</table>
Operation

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 1 0 0 0 0 0 0 0 0 0 1
    imm16
```

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);
AAArch64.CallSupervisor(imm16);
```
SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWPL and SWPAL store to memory with release semantics.
- SWP has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 1  | 0  | 0  | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

size
32-bit SWP (size == 10 & A == 0 & R == 0)

\[
\text{SWP} \ <\text{Ws}> , \ <\text{Wt}> , \ [<\text{Xn}|\text{SP}>]
\]

32-bit SWPA (size == 10 & A == 1 & R == 0)

\[
\text{SWPA} \ <\text{Ws}> , \ <\text{Wt}> , \ [<\text{Xn}|\text{SP}>]
\]

32-bit SWPAL (size == 10 & A == 1 & R == 1)

\[
\text{SWPAL} \ <\text{Ws}> , \ <\text{Wt}> , \ [<\text{Xn}|\text{SP}>]
\]

32-bit SWPL (size == 10 & A == 0 & R == 1)

\[
\text{SWPL} \ <\text{Ws}> , \ <\text{Wt}> , \ [<\text{Xn}|\text{SP}>]
\]

64-bit SWP (size == 11 & A == 0 & R == 0)

\[
\text{SWP} \ <\text{Xs}> , \ <\text{Xt}> , \ [<\text{Xn}|\text{SP}>]
\]

64-bit SWPA (size == 11 & A == 1 & R == 0)

\[
\text{SWPA} \ <\text{Xs}> , \ <\text{Xt}> , \ [<\text{Xn}|\text{SP}>]
\]

64-bit SWPAL (size == 11 & A == 1 & R == 1)

\[
\text{SWPAL} \ <\text{Xs}> , \ <\text{Xt}> , \ [<\text{Xn}|\text{SP}>]
\]

64-bit SWPL (size == 11 & A == 0 & R == 1)

\[
\text{SWPL} \ <\text{Xs}> , \ <\text{Xt}> , \ [<\text{Xn}|\text{SP}>]
\]

\[
\text{if} \ \text{!HaveAtomicExt}() \ \text{then} \ \text{UNDEFINED};
\]

integer \( t = \text{UInt}(\text{Rt}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
integer \( s = \text{UInt}(\text{Rs}) \);

integer \( \text{datasize} = 8 \ll \text{UInt}(\text{size}); \)
integer \( \text{regsize} = \text{if} \ \text{datasize} == 64 \ \text{then} \ 64 \ \text{else} \ 32; \)
\[
\text{AccType} \ \text{ldacctype} = \text{if} \ A == '{1}' \ \&\& \ \text{Rt} != '{11111}' \ \text{then} \ \text{AccType\_ORDEREDATOMICRW} \ \text{else} \ \text{AccType\_ATOMICRW};
\]
\[
\text{AccType} \ \text{stacctype} = \text{if} \ R == '{1}' \ \text{then} \ \text{AccType\_ORDEREDATOMICRW} \ \text{else} \ \text{AccType\_ATOMICRW};
\]
boolean \( \text{tag\_checked} = n != 31; \)

Assembler Symbols

\[
<\text{Ws}> \quad \text{Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.}
\]
\[
<\text{Wt}> \quad \text{Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.}
\]
\[
<\text{Xs}> \quad \text{Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.}
\]
\[
<\text{Xt}> \quad \text{Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.}
\]
\[
<\text{Xn}|\text{SP}> \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.}
\]
Operation

bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, regsize);
SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

**Integer**
(Armv8.1)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0 A R 1</td>
</tr>
</tbody>
</table>

**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;
bits(8) data;
bits(8) store_value;

if \texttt{HaveMTEExt()} then
  \texttt{SetTagCheckedInstruction}(tag\_checked);

if \(n = 31\) then
  \texttt{CheckSPAlignment}();
  address = \texttt{SP}[];
else
  address = \texttt{X}[n];

store\_value = \texttt{X}[s];
data = \texttt{MemAtomic}(address, \texttt{MemAtomicOp\_SWP},\texttt{store\_value}, ldacctype, stacctype);
\texttt{X}[t] = \texttt{ZeroExtend}(data, 32);
SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

### Integer
(Armv8.1)

<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>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size

**SWPAH (A == 1 && R == 0)**

SwPAH <Ws>, <Wt>, [<Xn|SP>]

**SWPALH (A == 1 && R == 1)**

SwPALH <Ws>, <Wt>, [<Xn|SP>]

**SWPH (A == 0 && R == 0)**

SwPH <Ws>, <Wt>, [<Xn|SP>]

**SWPLH (A == 0 && R == 1)**

SwPLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(16) data;
bits(16) store_value;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);
SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| sf | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | N | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Rn | Rd |

32-bit (sf == 0 && N == 0)

SXTB <Wd>, <Wn>

is equivalent to

SBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

SXTB <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of **SBFM**. This means:

- The encodings in this description are named to match the encodings of **SBFM**.
- The description of **SBFM** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>SF</th>
<th>IMM</th>
<th>N</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0)

**SXTH** `<Wd>, <Wn>`

is equivalent to

**SBFM** `<Wd>, <Wn>, #0, #15`

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

**SXTH** `<Xd>, <Wn>`

is equivalent to

**SBFM** `<Xd>, <Xn>, #0, #15`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of **SBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | Rn  | Rd |

sf opc N immr imms

64-bit

SXTW <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #31

and is always the preferred disassembly.

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
System instruction. For more information, see Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions for the encodings of System instructions.

This instruction is used by the aliases AT, CFP, CPP, DC, DVP, IC, and TLBI.

<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>1</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);

### Assembler Symbols

- **<op1>** Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- **<Cn>** Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- **<Cm>** Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- **<op2>** Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- **<Xt>** Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>CRn == '0111' &amp;&amp; CRm == '100x' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_AT</td>
</tr>
<tr>
<td>CFP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '100'</td>
</tr>
<tr>
<td>CPP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '111'</td>
</tr>
<tr>
<td>DC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_DC</td>
</tr>
<tr>
<td>DVP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '101'</td>
</tr>
<tr>
<td>IC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_IC</td>
</tr>
<tr>
<td>TLBI</td>
<td>CRn == '1000' &amp;&amp; SysOp(op1,'1000',CRm,op2) == Sys_TLBI</td>
</tr>
</tbody>
</table>

### Operation

AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
System instruction with result. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | op1 | CRn | CRm | op2 | Rt |

YSYL `<Xt>`, `#<op1>`, `<Cn>`, `<Cm>`, `#<op2>`

`AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);`

```c
integer t = UInt(Rt);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys.crm = UInt(CRm);
```

**Assembler Symbols**

- `<Xt>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
- `<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- `<Cn>` Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- `<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- `<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

**Operation**

```c
// No architecturally defined instructions here.
X[t] = AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys.crm, sys_op2);
```
TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

```
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

- `<R>` Is a width specifier, encoded in “b5”:
  - b5  <R>
    - 0    W
    - 1    X
  - In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

- `<t>` Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

- `<imm>` Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

- `<label>` Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];
if operand<bit_pos> == op then
  BranchTo(PC[] + offset, BranchType_DIR);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| b5 | 0  | 1  | 1  | 0  | 1  | 1  | 0  |
| op |

TBZ <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**

- \(<R>\) Is a width specifier, encoded in “b5”:
  
  \[ \begin{array}{|c|c|}
  \hline
  b5 & \text{<R>} \\
  \hline
  0 & W \\
  1 & X \\
  \hline
  \end{array} \]

  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 \([0\text{ to }63]\), encoded in "b5:b40".

- <label> Is the program label to be conditionally branched to. It's 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 © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
TLBI

TLB Invalidate operation. For more information, see \textit{op0==0b01, cache maintenance, TLB maintenance, and address translation instructions}.

This is an alias of \texttt{SYS}. This means:

- The encodings in this description are named to match the encodings of \texttt{SYS}.
- The description of \texttt{SYS} gives the operational pseudocode for this instruction.

\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 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0
   L CRm op1 CRn op2 Rt
\end{verbatim}

TLBI \texttt{<tlbi_op>\{, <Xt>\}}

is equivalent to

\begin{verbatim}
   SYS #<op1>, C8, <Cm>, #<op2>\{, <Xt>\}
\end{verbatim}

and is the preferred disassembly when \texttt{SysOp(op1,’1000’,CRm,op2) == Sys_TLBI}.

\textbf{Assembler Symbols}

\begin{itemize}
  \item \texttt{<op1>} \hspace{1cm} Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
  \item \texttt{<Cm>} \hspace{1cm} Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
  \item \texttt{<op2>} \hspace{1cm} Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
  \item \texttt{<tlbi_op>} \hspace{1cm} Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in “op1:CRm:op2”:
\end{itemize}
<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>&lt;tbi_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0001</td>
<td>000</td>
<td>VMALLE10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>001</td>
<td>VAE10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>010</td>
<td>ASIDE10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>011</td>
<td>VAAE10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>101</td>
<td>VALE10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>001</td>
<td>RVAE11S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>011</td>
<td>RVAAE11S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>101</td>
<td>RVALE11S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>111</td>
<td>RVAALE11S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>000</td>
<td>VMALLE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>001</td>
<td>VAE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>010</td>
<td>ASIDE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>011</td>
<td>VAAE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>101</td>
<td>VALE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>111</td>
<td>VAALE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>001</td>
<td>RVAE10S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>011</td>
<td>RVAAE10S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>101</td>
<td>RVALE10S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>001</td>
<td>RVAE1</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>011</td>
<td>RVAAE1</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>101</td>
<td>RVALE1</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>111</td>
<td>RVAALE1</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>000</td>
<td>VMALLE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>001</td>
<td>VAE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>010</td>
<td>ASIDE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>011</td>
<td>VAAE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>101</td>
<td>VALE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>111</td>
<td>VAALE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>001</td>
<td>IPAS2E11S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>010</td>
<td>RIPAS2E11S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>101</td>
<td>IPAS2LE11S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>110</td>
<td>RIPAS2LE11S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>000</td>
<td>ALLE20S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>001</td>
<td>VAE20S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>100</td>
<td>ALLE10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>101</td>
<td>VALE20S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>110</td>
<td>VMALLS12E10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>100</td>
<td>0010</td>
<td>001</td>
<td>RVAE21S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0010</td>
<td>101</td>
<td>RVALE21S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>000</td>
<td>ALLE21S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>001</td>
<td>VAE21S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>100</td>
<td>ALLE11S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>101</td>
<td>VALE21S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>110</td>
<td>VMALLS12E11S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>000</td>
<td>IPAS2E10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>001</td>
<td>IPAS2E1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>010</td>
<td>RIPAS2E1</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>011</td>
<td>RIPAS2E10S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>100</td>
<td>IPAS2LE10S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>101</td>
<td>IPAS2LE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>110</td>
<td>RIPAS2LE1</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>111</td>
<td>RIPAS2LE10S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0101</td>
<td>001</td>
<td>RVAE20S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0101</td>
<td>101</td>
<td>RVALE20S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0110</td>
<td>001</td>
<td>RVAE2</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0110</td>
<td>101</td>
<td>RVALE2</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>000</td>
<td>ALLE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>001</td>
<td>VAE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>100</td>
<td>ALLE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>101</td>
<td>VALE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>110</td>
<td>VMALLS12E1</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>000</td>
<td>ALLE30S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>001</td>
<td>VAE30S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>101</td>
<td>VALE30S</td>
<td>FEAT_TLBIOS</td>
</tr>
<tr>
<td>110</td>
<td>0010</td>
<td>001</td>
<td>RVAE31S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0010</td>
<td>101</td>
<td>RVALE31S</td>
<td>FEAT_TLBIIRANGE</td>
</tr>
<tr>
<td>op1</td>
<td>CRm</td>
<td>op2</td>
<td>&lt;tlbi_op&gt;</td>
<td>Architectural Feature</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>000</td>
<td>ALLE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>001</td>
<td>VAE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>101</td>
<td>VALE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0101</td>
<td>001</td>
<td>RVAE30S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0101</td>
<td>101</td>
<td>RVALE30S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>001</td>
<td>RVAE3</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>101</td>
<td>RVALE3</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>000</td>
<td>ALLE3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>001</td>
<td>VAE3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>101</td>
<td>VALE3</td>
<td>-</td>
</tr>
</tbody>
</table>

<Xt>  Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

**Operation**

The description of SYS gives the operational pseudocode for this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**TSB CSYNC**

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If `FEAT_TRF` is not implemented, this instruction executes as a NOP.

**System**

(Armv8.4)

```
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 1 0 0 1 0 0 1 0 1 1 1 1 1
```

**Operation**

```
if !HaveSelfHostedTrace() then EndOfInstruction();

TraceSynchronizationBarrier();
```
TST (immediate)

Test bits (immediate), setting the condition flags and discarding the result: \( Rn \ AND \ imm \).

This is an alias of ANDS (immediate). This means:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccccccccccc}
\hline
\sf & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & N & \ immr & \ imms & \ Rn & 1 & 1 & 1 & 1 & 1 & \ Rd \\
opc
\end{array}
\]

### 32-bit (sf == 0 \&\& N == 0)

TST <Wn>, #:imm

is equivalent to

ANDS WZR, <Wn>, #:imm

and is always the preferred disassembly.

### 64-bit (sf == 1)

TST <Xn>, #:imm

is equivalent to

ANDS XZR, <Xn>, #:imm

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<imm>` For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Operation**

The description of ANDS (immediate) gives the operational pseudocode for this instruction.
TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ANDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

TST $<Wn>, <Wm>{, <shift> #<amount>}

is equivalent to

ANDS WZR, $<Wn>, <Wm>{, <shift> #<amount>}

and is always the preferred disassembly.

64-bit (sf == 1)

TST $<Xn>, <Xm>{, <shift> #<amount>}

is equivalent to

ANDS XZR, $<Xn>, <Xm>{, <shift> #<amount>}

and is always the preferred disassembly.

Assembler Symbols

$<Wn>$ Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

$<Wm>$ Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

$<Xn>$ Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

$<Xm>$ Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

$<shift>$ Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

$<amount>$ For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UBFIZ

Unsigned Bitfield Insert in Zeros copies a bitfield of \(<width>\) bits from the least significant bits of the source register to bit position \(<lsb>\) of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

```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
sf 1 0 1 0 0 1 1 0  N  immr  imms  Rn  Rd
opc
```

### 32-bit (sf == 0 && N == 0)

UBFIZ \(<Wd>, <Wn>, #<lsb>, #<width>\)

is equivalent to

UBFM \(<Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)\)

and is the preferred disassembly when UInt(imms) < UInt(immr).

### 64-bit (sf == 1 && N == 1)

UBFIZ \(<Xd>, <Xn>, #<lsb>, #<width>\)

is equivalent to

UBFM \(<Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)\)

and is the preferred disassembly when UInt(imms) < UInt(immr).

#### Assembler Symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<lsb>\) For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- \(<width>\) For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

#### Operation

The description of UBFM gives the operational pseudocode for this instruction.

#### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UBFM**

Unsigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly. If `<imms>` is greater than or equal to `<immr>`, this copies a bitfield of `<imms>-<immr>+1` bits starting from bit position `<immr>` in the source register to the least significant bits of the destination register. If `<imms>` is less than `<immr>`, this copies a bitfield of `<imms>+1` bits from the least significant bits of the source register to bit position `(regsize-<immr>)` of the destination register, where `regsize` is the destination register size of 32 or 64 bits. In both cases the destination bits above and below the bitfield are set to zero.

This instruction is used by the aliases `LSL (immediate)`, `LSR (immediate)`, `UBFIZ`, `UBFX`, `UXTB`, and `UXTH`.

32-bit (sf == 0 & N == 0)

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit (sf == 1 & N == 1)

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
integer datasize = if sf == '1' then 64 else 32;
integer \( R \);
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;
\( R = \text{UInt}(immr) \);
\( (wmask, tmask) = \text{DecodeBitMasks}(N, imms, immr, FALSE) \);

**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><code>LSL (immediate)</code></td>
<td>32-bit</td>
<td><code>imms != '011111'</code> &amp; <code>imms + 1 == immr</code></td>
</tr>
<tr>
<td><code>LSL (immediate)</code></td>
<td>64-bit</td>
<td><code>imms != '111111'</code> &amp; <code>imms + 1 == immr</code></td>
</tr>
<tr>
<td><code>LSR (immediate)</code></td>
<td>32-bit</td>
<td><code>imms == '011111'</code></td>
</tr>
</tbody>
</table>
Alias | Of variant | Is preferred when
--- | --- | ---
LSR (immediate) | 64-bit | imms == '111111'
UBFIZ | UInt(imms) < UInt(immr) | 
UBFX | BFXPreferred(sf, opc<1>, imms, immr) | 
UXTB | immr == '000000' && imms == '000111' | 
UXTH | immr == '000000' && 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.
UBFX

Unsigned Bitfield Extract copies a bitfield of \(<width>\) bits starting from bit position \(<lsb>\) in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | N  | immr| imms| Rd |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

32-bit (sf == 0 && N == 0)

UBFX \(<Wd>, <Wn>, #<lsb>, #<width>\)

is equivalent to

UBFM \(<Wd>, <Wn>, #<lsb>, #(lsb)+<width>-1\)

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

64-bit (sf == 1 && N == 1)

UBFX \(<Xd>, <Xn>, #<lsb>, #<width>\)

is equivalent to

UBFM \(<Xd>, <Xn>, #<lsb>, #(lsb)+<width>-1\)

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

Assembler Symbols

\(<Wd>\)  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Wn>\)  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
\(<Xd>\)  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Xn>\)  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
\(<lsb>\)  For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
          For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
\(<width>\)  For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
            For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UDF

Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanently UNDEFINED in the Armv8-A architecture.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 imm16
UDF #<imm>

// The imm16 field is ignored by hardware.
UNDEFINED;
```

Assembler Symbols

<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

Operation

// No operation.
UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

| sf | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 1 | 0 | Rd | 01 |

32-bit (sf == 0)

UDIV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

UDIV <Xd>, <Xn>, < Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;
if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));
X[d] = result<datasize-1:0>;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UMADDL**

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias **UMULL**.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

UMADDL <Xd>, <Wn>, <Wm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

**Assembler Symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UMULL</strong></td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

\[
\begin{align*}
\text{bits}(32) \text{ operand1} &= X[n]; \\
\text{bits}(32) \text{ operand2} &= X[m]; \\
\text{bits}(64) \text{ operand3} &= X[a]; \\
\text{integer result;} \\
\text{result} &= \text{Int}(\text{operand3}, \text{TRUE}) + (\text{Int}(\text{operand1}, \text{TRUE}) \times \text{Int}(\text{operand2}, \text{TRUE})); \\
X[d] &= \text{result<63:0>};
\end{align*}
\]

**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.
UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of UMSUBL. This means:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | U  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 0  | Rm |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Rd |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

UMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

UMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMSUBL**

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias **UMNEGL**.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   |   | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |    | Rm | 1 |    | Ra |    | Rn |    | Rd |

**UMSUBL** <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>UMNEGL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
integer result;
result = \text{Int}(\text{operand3}, \text{TRUE}) - (\text{Int}(\text{operand1}, \text{TRUE}) * \text{Int}(\text{operand2}, \text{TRUE}));
Xd[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 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 0  | 0  | 1  | 1  | 1  | 1  | 0  | Rm | 0  | (1)|(1)|(1)|(1)|(1) | Rn | Rd |

UMULH <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;
result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d] = result<127:64>;

Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of UMADDL. This means:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccc}
\hline
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & Rm & 0 & 1 & 1 & 1 & 1 & Rn & Rd \\
\end{array}
\]

UMULL \(<Xd>, <Wn>, <Wm>\)

is equivalent to

UMADDL \(<Xd>, <Wn>, <Wm>, XZR\)

and is always the preferred disassembly.

Assembler Symbols

- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UXTB**

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of **UBFM**. This means:

- The encodings in this description are named to match the encodings of **UBFM**.
- The description of **UBFM** gives the operational pseudocode for this instruction.

![Register Bit Positions](image)

32-bit

**UXTB** `<Wd>, <Wn>`

is equivalent to

**UBFM** `<Wd>, <Wn>, #0, #7`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of **UBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

32-bit

UXTH <Wd>, <Wn>

is equivalent to

UBFM <Wd>, <Wn>, #0, #15

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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.

```
WFE
// Empty.

Operation

Hint_WFE(-1);
```
**WFET**

Wait For Event with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*. As described in *Wait For Event mechanism and Send event*, the execution of a WFET instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- **Traps to EL1 of EL0 execution of WFE and WFI instructions.**
- **Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.**
- **Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.**

**System**

(Armv8.7)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

WFET <Xt>

if !HaveFeatWFxT() then UNDEFINED;
integer d = UInt(Rd);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the “Rd” field.

Operation

```c
bits(64) operand = X[d];
integer localtimeout = UInt(operand);
if Halted() & ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();
Hint_WFE(localtimeout);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1
``` CRm op2

WFI

   // Empty.

Operation

```
Hint_WFI(-1);
```
WFIT

Wait For Interrupt with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see Wait For Interrupt. As described in Wait For Interrupt, the execution of a WFIT instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- Traps to EL1 of EL0 execution of WFE and WFI instructions.
- Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.
- Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.

System
(Armv8.7)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | R  |

```
WFIT <Xt>
if !HaveFeatWFxT() then UNDEFINED;
integer d = UInt(Rd);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

bits(64) operand = X[d];
integer localtimeout = UInt(operand);
if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();

Hint_WFI(localtimeout);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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)

```plaintext
if !HaveFlagFormatExt() then UNDEFINED;
```

Operation

```plaintext
bit N = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bz Z = PSTATE.Z AND PSTATE.C;
bz C = PSTATE.C OR PSTATE.Z;
bz V = NOT(PSTATE.C) AND PSTATE.Z;
PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = V;
```
XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPACI and XPACD, and is in LR for XPACLRI. The XPACD instruction is used for data addresses, and XPACI and XPACLRI are used for instruction addresses.

It has encodings from 2 classes: Integer and System

### Integer
(Armv8.3)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 1 1 0 0 0 0 1 0 1 0 0 0 D 1 1 1 1 1</td>
</tr>
<tr>
<td>Rn</td>
</tr>
</tbody>
</table>

#### XPACD (D == 1)

XPACD <Xd>

#### XPACI (D == 0)

XPACI <Xd>

```plaintext
test = (D == '1');
integer d = UInt(Rd);
if !HavePACExt() then
    UNDEFINED;
```

### System
(Armv8.3)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

XPACLRI

```plaintext
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

```plaintext
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*.

```
31 30 29 28 27 26 25 24 23 22 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 0 1 1 1 1 1
CRm        op2
```

**YIELD**

```
// Empty.
```

**Operation**

```
Hint_Yield();
```

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.
AESC: 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).
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).
FOR3: Three-way Exclusive OR.
EXT: Extract vector from pair of vectors.
FABD: Floating-point Absolute Difference (vector).
FABS (scalar): Floating-point Absolute value (scalar).
FABS (vector): Floating-point Absolute value (vector).
FACGE: Floating-point Absolute Compare Greater than or Equal (vector).
FACGT: Floating-point Absolute Compare Greater than (vector).
FADD (scalar): Floating-point Add (scalar).
FADD (vector): Floating-point Add (vector).
FADDP (scalar): Floating-point Add Pair of elements (scalar).
FADDP (vector): Floating-point Add Pairwise (vector).
FCADD: Floating-point Complex Add.
FCCMP: Floating-point Conditional quiet Compare (scalar).
FCCMPE: Floating-point Conditional signaling Compare (scalar).
FCMEQ (register): Floating-point Compare Equal (vector).
FCMEQ (zero): Floating-point Compare Equal to zero (vector).
FCMGE (register): Floating-point Compare Greater than or Equal (vector).
FCMGE (zero): Floating-point Compare Greater than or Equal to zero (vector).
FCMGTE (register): Floating-point Compare Greater than (vector).
FCMGTE (zero): Floating-point Compare Greater than zero (vector).
FCMLA: Floating-point Complex Multiply Accumulate.
FCMLA (by element): Floating-point Complex Multiply Accumulate (by element).
FCMLE (zero): Floating-point Compare Less than or Equal to zero (vector).
FCMUL (zero): Floating-point Compare Less than zero (vector).
FCMP: Floating-point quiet Compare (scalar).
FCMPE: Floating-point signaling Compare (scalar).
FCSEL: Floating-point Conditional Select (scalar).
FCVT: Floating-point Convert precision (scalar).
FCVTAS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).
FCVTAS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).
FCVTAU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).
FCVTAU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).
FCVTL, FCVTL2: Floating-point Convert to higher precision Long (vector).
FCVTMS (scalar): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).
FCVTMS (vector): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).
FCVTMU (scalar): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).
FCVTMU (vector): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).
FCVTN, FCVTN2: Floating-point Convert to lower precision Narrow (vector).
FCVTNS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).
FCVTNS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).
FCVTNU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).
FCVTNU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).
FCVTPS (scalar): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).
FCVTPS (vector): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).
FCVTPU (scalar): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).
FCVTPU (vector): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).
FCVTXN, FCVTXN2: Floating-point Convert to lower precision Narrow, rounding to odd (vector).
FCVTZS (scalar, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).
FCVTZS (scalar, integer): Floating-point Convert to Signed integer, rounding toward Zero (scalar).
FCVTZS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTZS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
FCVTZU (scalar, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).
FCVTZU (scalar, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).
FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).
FDIV (scalar): Floating-point Divide (scalar).
FDIV (vector): Floating-point Divide (vector).
FCVTZS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.
FMADD: Floating-point fused Multiply-Add (scalar).
FMAX (scalar): Floating-point Maximum (scalar).
FMAX (vector): Floating-point Maximum (vector).
FMAXNM (scalar): Floating-point Maximum Number (scalar).
FMAXNM (vector): Floating-point Maximum Number (vector).
FMAXNMP (scalar): Floating-point Maximum Number of Pair of elements (scalar).
FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).
FMAXNMV: Floating-point Maximum Number across Vector.

FMAXP (scalar): Floating-point Maximum of Pair of elements (scalar).

FMAXP (vector): Floating-point Maximum Pairwise (vector).

FMAX: 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).

FMINNMV: 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 Fused Multiply-Subtract (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).

FNNEG (scalar): Floating-point Negate (scalar).

FNNEG (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.
FRECXP: Floating-point Reciprocal exponent (scalar).
PRINT32X (scalar): Floating-point Round to 32-bit Integer, using current rounding mode (scalar).
PRINT32X (vector): Floating-point Round to 32-bit Integer, using current rounding mode (vector).
PRINT32Z (scalar): Floating-point Round to 32-bit Integer toward Zero (scalar).
PRINT32Z (vector): Floating-point Round to 32-bit Integer toward Zero (vector).
PRINT64X (scalar): Floating-point Round to 64-bit Integer, using current rounding mode (scalar).
PRINT64X (vector): Floating-point Round to 64-bit Integer, using current rounding mode (vector).
PRINT64Z (scalar): Floating-point Round to 64-bit Integer toward Zero (scalar).
PRINT64Z (vector): Floating-point Round to 64-bit Integer toward Zero (vector).
PRINTA (scalar): Floating-point Round to Integral, to nearest with ties to Away (scalar).
PRINTA (vector): Floating-point Round to Integral, to nearest with ties to Away (vector).
PRINTI (scalar): Floating-point Round to Integral, using current rounding mode (scalar).
PRINTI (vector): Floating-point Round to Integral, using current rounding mode (vector).
PRINTM (scalar): Floating-point Round to Integral, toward Minus infinity (scalar).
PRINTM (vector): Floating-point Round to Integral, toward Minus infinity (vector).
PRINTN (scalar): Floating-point Round to Integral, to nearest with ties to even (scalar).
PRINTN (vector): Floating-point Round to Integral, to nearest with ties to even (vector).
PRINTP (scalar): Floating-point Round to Integral, toward Plus infinity (scalar).
PRINTP (vector): Floating-point Round to Integral, toward Plus infinity (vector).
PRINTX (scalar): Floating-point Round to Integral exact, using current rounding mode (scalar).
PRINTX (vector): Floating-point Round to Integral exact, using current rounding mode (vector).
PRINTZ (scalar): Floating-point Round to Integral, toward Zero (scalar).
PRINTZ (vector): Floating-point Round to Integral, toward Zero (vector).
FRSRTS: Floating-point Reciprocal Square Root Estimate.
FRSRTS: 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.

**MOV (vector)**: Move vector: an alias of ORR (vector, register).

**MOVI**: Move Immediate (vector).

**MUL (by element)**: Multiply (vector, by element).

**MUL (vector)**: Multiply (vector).

**MVN**: Bitwise NOT (vector): an alias of NOT.

**MVNI**: Move inverted Immediate (vector).

**NEG (vector)**: Negate (vector).

**NOT**: Bitwise NOT (vector).

**ORN (vector)**: Bitwise inclusive OR NOT (vector).

**ORR (vector, immediate)**: Bitwise inclusive OR (vector, immediate).

**ORR (vector, register)**: Bitwise inclusive OR (vector, register).

**PMUL**: Polynomial Multiply.
**PMULL, PMULL2**: Polynomial Multiply Long.

**RADDHN, RADDHN2**: Rounding Add returning High Narrow.

**RAX1**: Rotate and Exclusive OR.

**RBIT (vector)**: Reverse Bit order (vector).

**REV16 (vector)**: Reverse elements in 16-bit halfwords (vector).

**REV32 (vector)**: Reverse elements in 32-bit words (vector).

**REV64**: Reverse elements in 64-bit doublewords (vector).

**RSHRN, RSHRN2**: Rounding Shift Right Narrow (immediate).

**RSUBHN, RSUBHN2**: Rounding Subtract returning High Narrow.

**SABA**: Signed Absolute difference and Accumulate.

**SABAL, SABAL2**: Signed Absolute difference and Accumulate Long.

**SABD**: Signed Absolute Difference.

**SABDL, SABDL2**: Signed Absolute Difference Long.

**SADALP**: Signed Add and Accumulate Long Pairwise.

**SADDL, SADDL2**: Signed Add Long (vector).

**SADDLP**: Signed Add Long Pairwise.

**SADDLV**: Signed Add Long across Vector.

**SADDDW, SADDDW2**: 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.
SMINY: Signed Minimum across Vector.
SMLSL, SMLSL2 (by element): Signed Multiply-Subtract Long (vector, by element).
SMLLA (vector): Signed 8-bit integer matrix multiply-accumulate (vector).
SMOV: Signed Move vector element to general-purpose register.
SOABS: Signed saturating Absolute value.
SOADD: Signed saturating Add.
SQDMLSL, SQDMLSL2 (by element): Signed saturating Doubling Multiply-Subtract Long (by element).
**A64 -- SIMD and Floating-point Instructions (alphabetic order)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQDMLSL, SQDMLSL2 (vector)</td>
<td>Signed saturating Doubling Multiply-Subtract Long.</td>
</tr>
<tr>
<td>SQDMULH (by element)</td>
<td>Signed saturating Doubling Multiply returning High half (by element).</td>
</tr>
<tr>
<td>SQDMULH (vector)</td>
<td>Signed saturating Doubling Multiply returning High half.</td>
</tr>
<tr>
<td>SQDMULL, SQDMULL2 (by element)</td>
<td>Signed saturating Doubling Multiply Long (by element).</td>
</tr>
<tr>
<td>SQDMULL, SQDMULL2 (vector)</td>
<td>Signed saturating Doubling Multiply Long.</td>
</tr>
<tr>
<td>SQNEG</td>
<td>Signed saturating Negate.</td>
</tr>
<tr>
<td>SORDMLAH (by element)</td>
<td>Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).</td>
</tr>
<tr>
<td>SORDMLAH (vector)</td>
<td>Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).</td>
</tr>
<tr>
<td>SORDMLSH (by element)</td>
<td>Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).</td>
</tr>
<tr>
<td>SORDMLSH (vector)</td>
<td>Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).</td>
</tr>
<tr>
<td>SORDMULH (by element)</td>
<td>Signed saturating Rounding Doubling Multiply returning High half (by element).</td>
</tr>
<tr>
<td>SORDMULH (vector)</td>
<td>Signed saturating Rounding Doubling Multiply returning High half.</td>
</tr>
<tr>
<td>SORSHL</td>
<td>Signed saturating Rounding Shift Left (register).</td>
</tr>
<tr>
<td>SORSHRN, SORSHRN2</td>
<td>Signed saturating Rounding Shift Right Narrow (immediate).</td>
</tr>
<tr>
<td>SORSHRUN, SORSHRUN2</td>
<td>Signed saturating Rounding Shift Right Unsigned Narrow (immediate).</td>
</tr>
<tr>
<td>SOSHL (immediate)</td>
<td>Signed saturating Shift Left (immediate).</td>
</tr>
<tr>
<td>SOSHL (register)</td>
<td>Signed saturating Shift Left (register).</td>
</tr>
<tr>
<td>SOSHLU</td>
<td>Signed saturating Shift Left Unsigned (immediate).</td>
</tr>
<tr>
<td>SOSHRN, SOSHUN2</td>
<td>Signed saturating Shift Right Narrow (immediate).</td>
</tr>
<tr>
<td>SOSHUN, SOSHUN2</td>
<td>Signed saturating Shift Right Unsigned Narrow (immediate).</td>
</tr>
<tr>
<td>SOSUB</td>
<td>Signed saturating Subtract.</td>
</tr>
<tr>
<td>SOXTN, SOXTN2</td>
<td>Signed saturating extract Narrow.</td>
</tr>
<tr>
<td>SOXTUN, SOXTUN2</td>
<td>Signed saturating extract Unsigned Narrow.</td>
</tr>
<tr>
<td>SRHADD</td>
<td>Signed Rounding Halving Add.</td>
</tr>
<tr>
<td>SRI</td>
<td>Shift Right and Insert (immediate).</td>
</tr>
<tr>
<td>SRSHL</td>
<td>Signed Rounding Shift Left (register).</td>
</tr>
<tr>
<td>SRSHDR</td>
<td>Signed Rounding Shift Right (immediate).</td>
</tr>
<tr>
<td>SRSRA</td>
<td>Signed Rounding Shift Right and Accumulate (immediate).</td>
</tr>
<tr>
<td>SSHL</td>
<td>Signed Shift Left (register).</td>
</tr>
<tr>
<td>SSHLL, SSHLL2</td>
<td>Signed Shift Left Long (immediate).</td>
</tr>
<tr>
<td>SSHR</td>
<td>Signed Shift Right (immediate).</td>
</tr>
<tr>
<td>SSRA</td>
<td>Signed Shift Right and Accumulate (immediate).</td>
</tr>
<tr>
<td>SSUBL, SSUBL2</td>
<td>Signed Subtract Long.</td>
</tr>
<tr>
<td>SSUBW, SSUBW2</td>
<td>Signed Subtract Wide.</td>
</tr>
<tr>
<td>ST1 (multiple structures)</td>
<td>Store multiple single-element structures from one, two, three, or four registers.</td>
</tr>
<tr>
<td>ST1 (single structure)</td>
<td>Store a single-element structure from one lane of one register.</td>
</tr>
</tbody>
</table>
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).

UADDLP: 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 (register).
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 hasencodingsfrom 2 classes: Scalar andVector

Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 0 | size | 1 0 0 0 0 | 0 1 0 1 1 1 0 | Rd | Rn
```

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

```
31 30 29 28 27 26 25 24 23 22 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 | size | 1 0 0 0 0 | 0 1 0 1 1 1 0 | Rd | Rn
```

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”:

```
<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":

ABS
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;I&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
for e = 0 to elements-1
   element = SInt(Elem[operand, e, esize]);
   if neg then
      element = -element;
   else
      element = Abs(element);
   Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
```

```
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 | 1 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
```

```
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**

```plaintext
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.

![Add returning High Narrow](image)

$\text{ADDHN(2)} <\text{Vd}.<\text{Tb}>, <\text{Vn}.<\text{Ta}>, <\text{Vm}$.<\text{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

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
b Gins(2*esize) element1;
b Gins(2*esize) element2;
b Gins(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              |

ADDP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize * 2;

Assembler Symbols

- <V> is the destination width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- <d> is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <T> is the source arrangement specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_ADD, operand, esize);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDP (vector)

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the \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}
ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\end{verbatim}

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;

Assembler Symbols

\begin{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

\begin{verbatim}
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;
\end{verbatim}

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>0</td>
</tr>
</tbody>
</table>

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;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_ADD, operand, esize);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESD

AES single round decryption.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D 1 0 0 1 1 1 0 0 0 1 0 0 0 1 0 1 1 0</td>
</tr>
</tbody>
</table>

AESD $<V_d>$.16B, $<V_n>$.16B

integer d = $\text{UInt}(R_d)$;
integer n = $\text{UInt}(R_n)$;
if !HaveAESExt() then UNDEFINED;

Assembler Symbols

$<V_d>$ Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

$<V_n>$ Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = $V[d]$;
bits(128) operand2 = $V[n]$;
bits(128) result;
result = operand1 EOR operand2;
result = AESInvSubBytes(AESInvShiftRows(result));
$V[d]$ = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESE

AES single round encryption.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  |

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.

\[
\begin{array}{cccccccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & Rn & Rd \\
\end{array}
\]

AESIMC \(<Vd>\.16B, <Vn>\.16B\)

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

Assembler Symbols

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\[\text{AArch64.CheckFPAdvSIMDEnabled}();\]

bits(128) operand = \(V[n]\);
bits(128) result;
result = \text{AESInvMixColumns}(\text{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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AESMC

AES mix columns.

<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>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>1</td>
<td>0</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>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

|  31 |  30 |  29 |  28 |  27 |  26 |  25 |  24 |  23 |  22 |  21 |  20 |  19 |  18 |  17 |  16 |  15 |  14 |  13 |  12 |  11 |  10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 0   | 1   | 1   | 0   | 0   | 0   | 1   | 1   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

RD | Rn | Rm | size

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

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
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.
BCAX

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register. This instruction is implemented only when `FEAT_SPA3` 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  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | Rn  |    |    |    |
| 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |

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

```plaintext
CheckFPAdvSIMDEnabled64();

bits(32) operand = V[n];
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

Elem [result, 0, 16] = FPConvertBF(operand, fpcr);
V[d] = result;
```

Copyright © 2010-2020 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 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  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | Rn | Rd |

**BFCVTN**(2) <Vd>, <Ta>, <Vn>.4S

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer part = UInt(Q);
integer elements = 64 DIV 16;

**Assembler Symbols**

<table>
<thead>
<tr>
<th></th>
<th>2</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 "Q":

<table>
<thead>
<tr>
<th></th>
<th>Ta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Ta</td>
</tr>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```assembly
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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**BFDOT (by element)**

BFloat16 floating-point dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the specified pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element of the destination vector that aligns with the pair of BF16 values in the first source vector. The instruction ignores the **FPCR** and does not update the **FPSR** exception status.

The BF16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive. **ID_AA64ISAR1_EL1** BF16 indicates whether this instruction is supported.

**Vector**

(Armv8.6)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | L  | M  | Rm | 1  | 1  | 1  | 1  | H  | 0  | Rn | Rd |

**BF DOT** **<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H[<index>]**

if !**HaveBF16Ext**() then UNDEFINED;
integer n = **UInt**(Rn);
integer m = **UInt**(M:Rm);
integer d = **UInt**(Rd);
integer i = **UInt**(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Ta>** Is an arrangement specifier, encoded in “Q”:
  
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Tb>** Is an arrangement specifier, encoded in “Q”:
  
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- **<index>** Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the "H:L" fields.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = **V**[n];
bits(128) operand2 = **V**[m];
bits(datasize) operand3 = **V**[d];
bits(datasize) result;

for e = 0 to elements-1

  bits(16) elt1_a = **Elem**[operand1, 2*e+0, 16];
  bits(16) elt1_b = **Elem**[operand1, 2*e+1, 16];
  bits(16) elt2_a = **Elem**[operand2, 2*i+0, 16];
  bits(16) elt2_b = **Elem**[operand2, 2*i+1, 16];

  bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
  **Elem**[result, e, 32] = BFAdd(**Elem**[operand3, e, 32], sum);

**V**[d] = result;

BFDOT (by element)
**BFDOT (vector)**

BFloat16 floating-point dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element of the destination vector that aligns with the pair of BF16 values in the first source vector. The instruction ignores the **FPCR** and does not update the **FPSR** exception status.

### Vector (Armv8.6)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 1 1 1 1</td>
</tr>
</tbody>
</table>

**BFDOT** `<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>`

if `!HaveBF16Ext()` then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>` Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;Ta&gt;</code></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><code>&lt;Tb&gt;</code></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 "Rm" field.

### Operation

```assembly
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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

<table>
<thead>
<tr>
<th>Vd</th>
<th>Vn</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>M</td>
<td>L</td>
</tr>
</tbody>
</table>

BFMLALB <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>0</th>
<th>B</th>
</tr>
</thead>
<tbody>
<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) result;
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(32) element2 = Elem[operand2, index, 16]:Zeros(16);
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2*e+sel, 16]:Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMLAL(operand1, element1, element2, FPCR[]);
V[d] = result;

Copyright © 2010-2020 Arm Limited. All rights reserved. This document is Non-Confidential.
BFMLALB, BFMLALT (vector)

BFLOAT16 floating-point widening multiply-add long (vector) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from BFLOAT16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector. This performs a fused multiply-add without intermediate rounding that honors all of the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR. ID_AA64ISAR1_EL1. BF16 indicates whether 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | Rm | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   |
| Rm | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   |
| 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | Rd | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   |

BFMLAL &lt;bt&gt; &lt;Vd&gt;.4S, &lt;Vn&gt;.8H, &lt;Vm&gt;.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

&lt;bt&gt; Is the bottom or top element specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;bt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
</tbody>
</table>

&lt;Vd&gt; Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

&lt;Vn&gt; Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

&lt;Vm&gt; Is the name of the second SIMD&FP source register, 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] = BFMulAdd(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)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

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);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

31 30 29 28 27 26 25 24 23 22 21 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 a b c x x x 1 0 1 d e f g h Rd
  op          cmode

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**

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.

\[
\begin{array}{cccccccccccccccc}
Q & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & Rm & 0 & 0 & 0 & 1 & 1 & 1 & 1 & Rn & Rd & size
\end{array}
\]

BIC \(<Vd>.<T>, <Vn>.<T>, <Vm>.<T>\)

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assemble 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

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\begin{align*}
\text{bits(datasize)} & \text{ operand1} = V[n]; \\
\text{bits(datasize)} & \text{ operand2} = V[m]; \\
\text{bits(datasize)} & \text{ result}; \\
\text{operand2} & = \text{NOT(operand2)}; \\
\text{result} & = \text{operand1 AND operand2}; \\
V[d] & = \text{result};
\end{align*}
\]

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.
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.

```
0 | Q | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
   | Rm | 0 | 0 | 0 | 1 | 1 | 1 |
   | Rn | Rd 
```

BIF <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);
type m = UInt(Rm);
type 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":
  ```plaintext
  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;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];
operand1 = V[d];
operand3 = NOT(V[m]);
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BIT

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 0 1 0 0 1 1 1 0 0 0 1 1 1

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

```c
CheckFPAdvSIMDEnabled64();
bots(datasize) operand1;
bots(datasize) operand3;
bots(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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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</td>
</tr>
</tbody>
</table>

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.

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

**U**

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & Rn & Rd
\end{array}
\]

\[
\text{CLS } <Vd>.,<T>,<Vn>.<T>
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{if } \text{size} == \text{'11'} \text{ then UNDEFINED;} \\
\text{integer } \text{esize} &= 8 \ll \text{UInt}(\text{size}); \\
\text{integer } \text{datasize} &= \text{if } Q == \text{'1'} \text{ then 128 else 64;} \\
\text{integer } \text{elements} &= \text{datasize} \div \text{esize};
\end{align*}
\]

\[
\text{CountOp } \text{countop} = \text{if } U == \text{'1'} \text{ then CountOp_CLZ else CountOp_CLS;}
\]

**Assembler Symbols**

\[
\begin{array}{c|c|c}
\text{size} & \text{Q} & <T> \\
\hline
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & \text{RESERVED}
\end{array}
\]

\[
\text{<Vn>} \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}
\]

\[
\text{<T>} \quad \text{Is an arrangement specifier, encoded in "size:Q":}
\]

**Operation**

\[
\text{CheckFPAdvSIMDEnabled64();}
\]

\[
\begin{align*}
\text{bits(datasize)} \text{ operand} &= V[n]; \\
\text{bits(datasize)} \text{ result}; \\
\end{align*}
\]

\[
\begin{align*}
\text{integer } \text{count}; \\
\text{for } e = 0 \text{ to elements-1} \\
\text{if } \text{countop} == \text{CountOp_CLS} \text{ then} \\
\quad \text{count} &= \text{CountLeadingSignBits} (\text{Elem}[\text{operand}, e, \text{esize}]); \\
\text{else} \\
\quad \text{count} &= \text{CountLeadingZeroBits} (\text{Elem}[\text{operand}, e, \text{esize}]); \\
\text{Elem}[\text{result}, e, \text{esize}] &= \text{count}<\text{esize}-1:0>;; \\
\text{V}[d] &= \text{result};
\end{align*}
\]

**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.
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  | 0  | 1  | 1  | 1  | 0  | size| 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |
| U  | Rd | Rn |
```

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;

integer count;
for e = 0 to elements-1
  if countop == CountOp_CLS then
    count = CountLeadingSignBits(Elem[operand, e, esize]);
  else
    count = CountLeadingZeroBits(Elem[operand, e, esize]);
  Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
0 1 1 1 1 1 1 0  | size 1 | Rm 1 0 0 0 1 1 | Rn | Rd
```

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

```
0 1 0 1 1 1 0 1  | size 1 | Rm 1 0 0 0 1 1 | Rn | Rd
```

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”:

| <V> | size  
<table>
<thead>
<tr>
<th></th>
<th></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>

<\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(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMEQ (zero)

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
31 30 29 28 27 26 25 24 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 0 size 1 0 0 0 0 0 1 0 0 1 1 1 1 Rd
op
```

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
U 0 1 0 0 1 1 1 1 size 1 0 0 0 0 0 1 0 0 1 1 1 1 Rd
op
```

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

<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>

<Vn>  
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT  test_passed = element > 0;
        when CompareOp_GE  test_passed = element >= 0;
        when CompareOp_EQ  test_passed = element == 0;
        when CompareOp_LE  test_passed = element <= 0;
        when CompareOp_LT  test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>1</td>
<td>Rm</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0</td>
<td>1</td>
<td>Rm</td>
</tr>
</tbody>
</table>

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>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();
bias(datasize) operand1 = V[n];
bias(datasize) operand2 = V[m];
bias(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 1 1 1 1 1 1 0 | size | 1 0 0 0 0 | 0 1 0 0 | 0 1 0 | Rn | Rd |

CMGE <V><d>, <V><n>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 | Q | 1 0 1 1 1 0 | size | 1 0 0 0 0 | 0 1 0 0 | 0 1 0 | Rn | Rd |

CMGE <Vd>.<T>, <Vn>.<T>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

```
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>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</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;
ingTEGER element;
boolean test_passed;

for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
  case comparison of
    when CompareOp_GT test_passed = element > 0;
    when CompareOp_GE test_passed = element >= 0;
    when CompareOp_EQ test_passed = element == 0;
    when CompareOp_LE test_passed = element <= 0;
    when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
  V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the \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}.

**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  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| U  | eq| Rm |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

CMGT \(<V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

**Vector**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>eq</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMGT \(<Vd>..<T>, <Vn>..<T>, <Vm>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

**Assembler Symbols**

\(<V>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<m>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in "size:Q".
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{n}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<\text{m}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
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}

### 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 0             | 0 1 0 1 0             | Rd                  |                     |
| 1 0 0 0 0             |                      |                      |                      |                      |                      |                     |
| Rn                   | op                  |                      |                      |                      |                      |                     |
```

CMGT \texttt{<V><d>, <V><n>, #0}

integer \( d = \texttt{UInt(Rd)}; \)
integer \( n = \texttt{UInt(Rn)}; \)

if size \(!=\) \texttt{''11''} then UNDEFINED;
integer esize = 8 \(<=\) \texttt{UInt(size)};
integer datasize = esize;
integer elements = 1;

\texttt{CompareOp} comparison;
\texttt{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

```

| 31 30 29 28 27 26 25 24 23 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rd |
| U | op |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |                      |
```

CMGT \texttt{<Vd>.<T>, <Vn>.<T>, #0}

integer \( d = \texttt{UInt(Rd)}; \)
integer \( n = \texttt{UInt(Rn)}; \)

if size:Q \(!=\) \texttt{''110''} then UNDEFINED;
integer esize = 8 \(<=\) \texttt{UInt(size)};
integer datasize = if Q \(!=\) \texttt{''1''} then 128 else 64;
integer elements = datasize DIV esize;

\texttt{CompareOp} comparison;
\texttt{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};

### Assembler Symbols

\texttt{<V> Is a width specifier, encoded in "size";}
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
```
```
bits(datasize) operand = V[n];
b ticks(datasize) result;
integer element;
boolean test_passed;
```
```
for e = 0 to elements-1
  element = $int(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 \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}

**Scalar**

```plaintext
0  1  1  1  1  1  1  0  |  size  |  Rd  0  0  1  1  0  1  |  Rn  Rd
U   eq

\textbf{CMHI} \texttt{<d>}, \texttt{<n>}, \texttt{<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**

```plaintext
0  1  0  1  1  1  1  0  |  size  |  Rd  0  0  1  1  0  1  |  Rn  Rd
U   eq

\textbf{CMHI} \texttt{<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**

- \texttt{<V>} Is a width specifier, encoded in “size”:

```
\begin{tabular}{|c|c|}
\hline
\texttt{size} & \texttt{<V>} \\
\hline
0x & RESERVED \\
10 & RESERVED \\
11 & D \\
\hline
\end{tabular}
```

- \texttt{<d>} Is the number of the SIMD&FP destination register, in the "Rd" field.
- \texttt{<n>} Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<m>} Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<T>} Is an arrangement specifier, encoded in “size:Q”:
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>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}> 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**

```pseudo
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.
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

```
0 1 1 1 1 1 0 | size | 1  
 0 0 1 1 1 1 | Rd |
   eq
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
in teger esize = 8 << UInt(size);
in teger datasize = esize;
in teger elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Vector

```
0 1 1 1 1 0 | size | 1  
 0 0 1 1 1 1 | Rd |
   eq
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
in teger esize = 8 << UInt(size);
in teger datasize = if Q == '1' then 128 else 64;
in teger elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**CMLE (zero)**

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U  op

CMLE <V><d>, <V><n>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U  op

CMLE <Vd>.<T>, <Vn>.<T>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Assembler Symbols

`<V>` Is a width specifier, encoded in “size”:
<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];
b bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
  case comparison of
    when CompareOp_GT test_passed = element > 0;
    when CompareOp_GE test_passed = element >= 0;
    when CompareOp_EQ test_passed = element == 0;
    when CompareOp_LE test_passed = element <= 0;
    when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the \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}

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 0 0 0 0 1 0 1 0 1 0</th>
<th>Rd</th>
</tr>
</thead>
</table>

\begin{verbatim}
CMLT <V><d>, <V><n>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
\end{verbatim}

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 0 0 0 0 1 0 1 0 1 0</th>
<th>Rd</th>
</tr>
</thead>
</table>

\begin{verbatim}
CMLT <Vd>..<T>, <Vn>..<T>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
\end{verbatim}

### Assembler Symbols

\begin{verbatim}
<V> Is a width specifier, encoded in "size":
\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{size} & \textbf{<V>} \\
\hline
0x & RESERVED \\
10 & RESERVED \\
11 & D \\
\hline
\end{tabular}
\end{center}
\end{verbatim}

\textit{<d>} Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\textit{<n>} Is the number of the SIMD&FP source register, encoded in the "Rn" field.

\textit{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\textit{<T>} Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
            Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMTST

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| 0 1 0 1 1 1 1 0 | size | 1 | 0 0 0 1 1 | Rm | 0 0 1 1 | Rn | Rd |
```

CMTST <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| 0 0 1 1 1 1 0 | size | 1 | 0 0 0 1 1 | Rm | 0 0 1 1 | Rn | Rd |
```

CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt; Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>0x</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q";
<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.
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 \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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

\texttt{CNT <Vd>.<T>, <Vn>.<T>}

\begin{verbatim}
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;
\end{verbatim}

\textbf{Assembler Symbols}

\begin{itemize}
  \item \texttt{<Vd>} is the name of the SIMD&FP destination register, encoded in the "Rd" field.
  \item \texttt{<T>} is an arrangement specifier, encoded in "size:Q":
  \begin{tabular}{|c|c|c|}
    \hline
    size & Q & <T> \\
    \hline
    00 & 0 & 8B \\
    00 & 1 & 16B \\
    01 & x & RESERVED \\
    1x & x & RESERVED \\
    \hline
  \end{tabular}
  \item \texttt{<Vn>} is the name of the SIMD&FP source register, encoded in the "Rn" field.
\end{itemize}

\textbf{Operation}

\begin{verbatim}
CheckFPAdvSIMD Enabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer count;
for e = 0 to elements-1
  count = BitCount(Elem[operand, e, esize]);
  Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
\end{verbatim}

\textbf{Operational information}

If PSTATE.DIT is 1:
\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
  \item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}
**DUP (element)**

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias **MOV (scalar)**.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 1 0 1 1 1 1 0 0 0 0 | imm5 | 0 0 0 0 0 1 | Rn | Rd |

DUP `<V><d>, <Vn>.<T>[<index>]`

- integer d = `UInt(Rd)``
- integer n = `UInt(Rn)``

integer size = `LowestSetBit(imm5)`;
if size > 3 then UNDEFINED;

integer index = `UInt(imm5<4:size+1>)``
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;

### Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 | Q | 0 1 1 1 0 0 0 0 | imm5 | 0 0 0 0 0 1 | Rn | Rd |

DUP `<Vd>.<T>, <Vn>.<Ts>[<index>]`

- integer d = `UInt(Rd)``
- integer n = `UInt(Rn)``

integer size = `LowestSetBit(imm5)`;
if size > 3 then UNDEFINED;

integer index = `UInt(imm5<4:size+1>)``
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

### Assembler Symbols

**<T>**

For the scalar variant: is the element width specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx1</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

For the vector variant: is an arrangement specifier, encoded in “imm5:Q”:
**<Ts>**
Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx1</td>
<td>0</td>
<td>0B</td>
</tr>
<tr>
<td>xxx1</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>xxx10</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>xxx10</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<V>**
Is the destination width specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>1000</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>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx1</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>1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

**<d>**
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vd>**
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(datasize) result;
bits(esize) element;

    element = Elem[operand, index, esize];
    for e = 0 to elements-1
        Elem[result, e, esize] = element;
    V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|
| mm5 0 0 0 1 1 1 0 0 0 0    | imm5 0 0 0 1 1               | Rn                           |
| Rd                           |
```

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”:

```
<table>
<thead>
<tr>
<th>imm5</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>xxxx1</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>xxx10</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>xxx10</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
```

<\(R\)> Is the width specifier for the general-purpose source register, encoded in “imm5”:

```
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>
```

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```c
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.
Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register. This instruction is implemented only when `FEAT_SHA3` is implemented.

### Advanced SIMD

**Armv8.2**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | Rm | 0  | Ra |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

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**

```assembly
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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | Rm | 0 | 0 | 0 | 1 | 1 | 1 | Rd |

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":
  - 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

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.

EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if Q == '0' && imm4<3> == '1' then UNDEFINED;
ninteger datasize = if Q == '1' then 128 else 64;
ninteger position = UInt(imm4) << 3;

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.
<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();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi:lo;
V[d] = concat<position+datasize-1:position>;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision (Armv8.2)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| Rm              | Rn              | Rd              |
| 0 1 1 1 1 1 1 0 1 1 0 | 0 0 1 0 1 1      |                 |
```

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 |
|-----------------|-----------------|-----------------|
| Rm              | Rn              | Rd              |
| 0 1 1 1 1 1 1 0 1 sz 1 | 1 1 0 1 0 1 1      |                 |
```

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 |
|-----------------|-----------------|-----------------|
| Rm              | Rn              | Rd              |
| U 0 0 0 1 1 1 0 1 1 0 | 0 0 0 1 0 1 1      |                 |
```

U
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| Q   | 1  | 0  | 1  | 1  | 0  | 1  | sz | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rm  | 1  | 1  | 0  | 1  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FABD <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

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q <T>

| 0  | 4H  |
| 1  | 8H  |

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \( V[n] \);
bits(datasize) operand2 = \( V[m] \);

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRTypen fpcr = FPCR[];

boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then \( V[n] \) else Zeros();

for e = 0 to elements-1

\( \text{element1} = \text{Elem}[\text{operand1}, e, \text{esize}] \);
\( \text{element2} = \text{Elem}[\text{operand2}, e, \text{esize}] \);
\( \text{diff} = \text{FPSub}(\text{element1}, \text{element2}, \text{fpcr}) \);
\( \text{Elem}[\text{result}, e, \text{esize}] = \text{if abs then} \ \text{FPAbs}(\text{diff}) \ \text{else diff} \);

\( V[d] = \text{result} \);
**FABS (vector)**

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

### Half-precision (Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

U

**FABS <Vd>.<T>, <Vn>.<T>**

```plaintext
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 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

U

**FABS <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 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":
  ```plaintext
  Q   <T>
  0   4H
  1   8H
  ````
  For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
  ```plaintext
  sz  Q   <T>
  0  0   2S
  0  1   4S
  1  0   RESERVED
  1  1   2D
  ```
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\texttt{CheckFPAdvSIMDEnabled64();}\n\texttt{bits(datasize) operand = V[n];}\n\texttt{bits(datasize) result;}\n\texttt{bits(esize) element;}\n\texttt{for e = 0 to elements-1}\n\texttt{element = \texttt{Elem}[operand, e, esize];}\n\texttt{if neg then}\n\texttt{element = FPNeg(element);}\n\texttt{else}\n\texttt{element = FPAbs(element);}\n\texttt{\texttt{Elem}[result, e, esize] = element;\nV[d] = result;
FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

FABS (scalar)

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 esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bots(128) result = if merge then V[d] else Zeros();
bots(esize) operand = V[n];

Elem[result, 0, esize] = 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 0 1 0
U E Rm
0 0 1 0 1 1
ac Rn
Rd
```

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 0 0 sz 1
U E Rm
1 1 1 0 1 1
ac Rn
Rd
```
integer \( d = \text{UInt}(R_d) \);
integer \( n = \text{UInt}(R_n) \);
integer \( m = \text{UInt}(R_m) \);
integer esize = 32 \ll \text{UInt}(sz);
integer datasize = esize;
integer elements = 1;

\begin{verbatim}
integer esize = 32 \ll \text{UInt}(sz);
integer datasize = esize;
integer elements = 1;
\end{verbatim}

\begin{verbatim}
CompareOp cmp;
boolean abs;
\end{verbatim}

\begin{verbatim}
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;
\end{verbatim}

\begin{verbatim}
Vector half precision
\end{verbatim}

\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 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 1 | Rn | Rd |
|-----------------|---------------|---------------|---------------|
|                |               |               |               |
\end{verbatim}

\begin{verbatim}
Vector single-precision and double-precision
\end{verbatim}

\begin{verbatim}
if !\text{HaveFP16Ext}() then UNDEFINED;
\end{verbatim}

\begin{verbatim}
integer d = \text{UInt}(R_d);
integer n = \text{UInt}(R_n);
integer m = \text{UInt}(R_m);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp cmp;
boolean abs;
\end{verbatim}

\begin{verbatim}
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;
\end{verbatim}

\begin{verbatim}
Vector single-precision and double-precision
\end{verbatim}

\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 | sz | 1 | Rm | 1 | 1 | 1 | 0 | 1 | 1 | Rn | Rd |
|-----------------|---------------|---------------|---------------|
|                |               |               |               |
\end{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
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.
<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.

For the half-precision variant: an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<V> 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>
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(eseize) element1;
bits(eseize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  if abs then
    element1 = FPAbs(element1);
    element2 = FPAbs(element2);
  case cmp of
    when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
    when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
    when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;}
**FACGT**

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

### Scalar half precision (Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FACGT <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

### Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;
case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Vector half precision
(Armv8.2)

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;
case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Vector single-precision and double-precision
integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
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 \(elements = \text{datasize} \div \text{esize}\);
\text{CompareOp} \ cmp;
\text{boolean} \ abs;

\text{case } E:U:ac \ of
  when '000' \ cmp = \text{CompareOp_EQ}; \ abs = \text{FALSE};
  when '010' \ cmp = \text{CompareOp_GE}; \ abs = \text{FALSE};
  when '011' \ cmp = \text{CompareOp_GE}; \ abs = \text{TRUE};
  when '110' \ cmp = \text{CompareOp_GT}; \ abs = \text{FALSE};
  when '111' \ cmp = \text{CompareOp_GT}; \ abs = \text{TRUE};
  otherwise UNDEFINED;

\textbf{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":

\[
\begin{array}{c|c}
\text{sz} & \text{<V>} \\
\hline
0 & S \\
1 & D \\
\end{array}
\]

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.
\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
\(<m>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

\[
\begin{array}{c|c}
\text{Q} & \text{<T>} \\
\hline
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\[
\begin{array}{c|c|c|c}
\text{sz} & \text{Q} & \text{<T>} \\
\hline
0 & 0 & 25 \\
0 & 1 & 45 \\
1 & 0 & \text{RESERVED} \\
1 & 1 & 20 \\
\end{array}
\]

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(etype) element1;
bits(etype) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bias(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
    element1 = Ele[operand1, e, esize];
    element2 = Ele[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)

```
0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 1 | Rn | Rd |
U
```

FADD \(<Vd>.<T>, <Vn>.<T>, <Vm>.<T>\)

```plaintext
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

```
0 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |
U
```

FADD \(<Vd>.<T>, <Vn>.<T>, <Vm>.<T>\)

```plaintext
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":
  ```plaintext
  Q | <T>
  0 | 4H
  1 | 8H
  ```
  For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

---

FADD (vector)  Page 663
<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>

<\text{n}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | Rm | 0  | 0  | 1  | 0  | 1  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 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 esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypel fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem(result, 0, esize) = FPAdd(operand1, operand2, fpcr);
V[d] = result;
FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**
(ARMv8.2)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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   | 1   | 1   | 0   | 1   | 1   | 0   | Rn   |   |   |
| sz   |

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   | 1   | 1   | 0   | 0   | sz | 1   | 1   | 0   | 0   | 0   | 0   | 0   | 1   | 1   | 0   | Rn   |   |   |
| sz   |

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":

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 0 |
|   | H |
|   | RESERVED |

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 0 |
|   | S |
|   | 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":

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 0 |
|   | 2H |
|   | RESERVED |

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:
### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FADD, operand, esize);
```

---

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADD P (vector)

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</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>
</tbody>
</table>

FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>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>
</tbody>
</table>

FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

FADD P (vector)
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```cpp
CheckFPAdvSIMDEnabled64();
```

```cpp
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
```

```cpp
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[]);
```

```cpp
V[d] = result;
```
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

- Considering the complex number from the second source register on an Argand diagram, the number is
  rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in
either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point
exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

Vector
(Armv8.3)

<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>size</td>
<td>0</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>rot</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>
</tbody>
</table>

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

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \text{V}[n];
bits(datasize) operand2 = \text{V}[m];
bits(datasize) operand3 = \text{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 = \text{FPNeg(Elem[operand2, e*2+1, esize]);}
      element3 = \text{Elem[operand2, e*2, esize];}
    when '1'
      element1 = \text{Elem[operand2, e*2+1, esize];}
      element3 = \text{FPNeg(Elem[operand2, e*2, esize]);}
  \text{Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR[]);}
  \text{Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR[]);}

\text{V[d] = result;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE. {N, Z, C, V} flags. If the condition does not pass then the PSTATE. {N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is 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 |

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.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];

if ConditionHolds(cond) then
    flags = FPCmpare(operand1, operand2, FALSE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the **PSTATE** condition flags to N=0, Z=0, C=1, and V=1.
FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the \texttt{PSTATE}.\{N, Z, C, V\} flags. If the condition does not pass then the \texttt{PSTATE}.\{N, Z, C, V\} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in \texttt{FPCR}, the exception results in either a flag being set in \texttt{FPSR}, or a synchronous exception being generated. For more information, see Floating-point exception traps.

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}
 Half-precision (ftype == 11)
 (Armv8.2)

 FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>

 Single-precision (ftype == 00)

 FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>

 Double-precision (ftype == 01)

 FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>

 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;
 end

 bits(4) flags = nzcv;
\end{verbatim}

Assembler Symbols

\begin{verbatim}
 <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.
\end{verbatim}
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(cond) then
    flags = FPC ompare(operand1, operand2, TRUE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the `PSTATE` condition flags to N=0, Z=0, C=1, and V=1.
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)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 0 0 1 0 0 1</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

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_GT; 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 0 1 1 1 1 0 0</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
integer $m = \text{UInt}(Rm)$;
integer $esize = 32 << \text{UInt}(sz)$;
integer $datasize = esize$;
integer $elements = 1$;
\textbf{CompareOp} cmp;
\textbf{boolean} abs;
\textbf{case} E:U:ac of
  \textbf{when} '000' cmp = \textbf{CompareOp\_EQ}; abs = \text{FALSE};
  \textbf{when} '010' cmp = \textbf{CompareOp\_GE}; abs = \text{FALSE};
  \textbf{when} '011' cmp = \textbf{CompareOp\_GE}; abs = \text{TRUE};
  \textbf{when} '110' cmp = \textbf{CompareOp\_GT}; abs = \text{FALSE};
  \textbf{when} '111' cmp = \textbf{CompareOp\_GT}; abs = \text{TRUE};
\text{otherwise} UNDEFINED;

\textbf{Vector half precision}
\textbf{(Armv8.2)}

\begin{array}{ccccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 & 23 \\
\hline
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
\text{U} & \text{E} & \text{ac} & Rm & Rn & Rd
\end{array}

\textbf{FCMEQ} \langle Vd \rangle < \langle T \rangle, \langle Vn \rangle < \langle T \rangle, \langle Vm \rangle < \langle T \rangle

if \text{!HaveFP16Ext()} then UNDEFINED;

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
integer $m = \text{UInt}(Rm)$;
integer $esize = 16$;
integer $datasize = \text{if} \ Q \ \text{==} \ '1' \ \text{then} \ 128 \ \text{else} \ 64$;
integer $elements = \text{datasize} \ \text{DIV} \ esize$;
\textbf{CompareOp} cmp;
\textbf{boolean} abs;
\textbf{case} E:U:ac of
  \textbf{when} '000' cmp = \textbf{CompareOp\_EQ}; abs = \text{FALSE};
  \textbf{when} '010' cmp = \textbf{CompareOp\_GE}; abs = \text{FALSE};
  \textbf{when} '011' cmp = \textbf{CompareOp\_GE}; abs = \text{TRUE};
  \textbf{when} '110' cmp = \textbf{CompareOp\_GT}; abs = \text{FALSE};
  \textbf{when} '111' cmp = \textbf{CompareOp\_GT}; abs = \text{TRUE};
\text{otherwise} UNDEFINED;

\textbf{Vector single-precision and double-precision}

\begin{array}{ccccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 & 23 \\
\hline
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
\text{U} & \text{E} & \text{ac} & Rm & Rn & Rd
\end{array}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Assembly Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(ese) element1;
bits(ese) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
  element1 = Elem(operand1, e, esize);
  element2 = Elem(operand2, e, esize);
  if abs then
    element1 = FPAbs(element1);
    element2 = FPAbs(element2);
  case cmp of
    when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
    when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
    when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
  Elem(result, e, esize) = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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)

```
<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>1</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>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>
</tr>
<tr>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

FCMEQ <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

```scala
CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

FCMEQ <V<d>, <V<n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

```scala
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**

```
0 Q 0 0 1 1 1 0 | 1 1 1 1 0 0 0 1 1 0 | Rn | Rd
```

FCMEQ `<Vd>.<T>, <Vn>.<T>, #0.0`

if `!HaveFP16Ext()` then UNDEFINED;

integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

```plaintext
CompareOp comparison;
```

```plaintext
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

```
0 Q 0 0 1 1 1 0 | 1 0 0 0 0 0 1 1 0 | Rn | Rd
```

FCMEQ `<Vd>.<T>, <Vn>.<T>, #0.0`

integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << `UInt(sz)`;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

```plaintext
CompareOp comparison;
```

```plaintext
case op:U of
  when '00' comparison = `CompareOp_GT`;
  when '01' comparison = `CompareOp_GE`;
  when '10' comparison = `CompareOp_EQ`;
  when '11' comparison = `CompareOp_LE`;
```

### Assembler Symbols

- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<V>`: Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>`: Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the half-precision variant: is an arrangement specifier, encoded in “Q”:

FCMEQ (zero)
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>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>

\(<\text{Vn}\>)\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \text{V}[n];
bits(datasize) result;
bits(esize) zero = \text{FPZero}(\text{'}0\text{'});
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
  element = \text{Elem}[\text{operand}, e, esize];
  case comparison of
    when \text{CompareOp_GT} test_passed = \text{FPCompareGT}(element, zero, FPCR[]);
    when \text{CompareOp_GE} test_passed = \text{FPCompareGE}(element, zero, FPCR[]);
    when \text{CompareOp_EQ} test_passed = \text{FPCompareEQ}(element, zero, FPCR[]);
    when \text{CompareOp_LE} test_passed = \text{FPCompareGE}(zero, element, FPCR[]);
    when \text{CompareOp_LT} test_passed = \text{FPCompareGT}(zero, element, FPCR[]);
  \text{Elem}[result, e, esize] = if test_passed then \text{Ones()} else \text{Zeros()};
\text{V}[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR_EL1, CPTR_EL2, and CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: \textit{Scalar half precision}, \textit{Scalar single-precision and double-precision}, \textit{Vector half precision} and \textit{Vector single-precision and double-precision}

**Scalar half precision**

\textit{(Armv8.2)}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | Rm | 0  | 0  | 1  | 0  | 0  | 1  | Rn | Rd |

\textit{FCMGE <Hd>, <Hn>, <Hm>
}

if \textit{!HaveFP16Ext()} then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
\textit{CompareOp} cmp;
boolean abs;

\texttt{case E:U:ac of}

\texttt{when '000' cmp = CompareOp_EQ; abs = FALSE;}
\texttt{when '010' cmp = CompareOp_GE; abs = FALSE;}
\texttt{when '011' cmp = CompareOp_GE; abs = TRUE;}
\texttt{when '110' cmp = CompareOp_GT; abs = FALSE;}
\texttt{when '111' cmp = CompareOp_GT; abs = TRUE;}
\texttt{otherwise UNDEFINED;}

**Scalar single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 0  | 1  | Rn | Rd |

\textit{FCMGE (register) Page 684}


**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;
```

```bash
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>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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
FCMGE <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;
```

```bash
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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 1  | 0  | 1  | 1  | 1  | 0  | | 0  | 1  | Rm | 1  | 1  | 1  | 0  | 0  | 1  | Rn | Rd |
| U   | E  | sz |    |    |    |    |    | |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

FCMGE (register)
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 $datasize = \text{if } Q == '1' \text{ then } 128 \text{ else } 64$;
integer $elements = \text{datasize \ div } esize$;

$\text{CompareOp } cmp$;
boolean $abs$;

case $E:U:ac$ of
  when '000' $cmp = \text{CompareOp_EQ}$; $abs = \text{FALSE}$;
  when '010' $cmp = \text{CompareOp_GE}$; $abs = \text{FALSE}$;
  when '011' $cmp = \text{CompareOp_GE}$; $abs = \text{TRUE}$;
  when '110' $cmp = \text{CompareOp_GT}$; $abs = \text{FALSE}$;
  when '111' $cmp = \text{CompareOp_GT}$; $abs = \text{TRUE}$;
  otherwise UNDEFINED;

Assembler Symbols

$<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
$<Hn>$ Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
$<Hm>$ Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

$<V>$ Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>$sz$</th>
<th>$&lt;V&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

$<d>$ Is the number of the SIMD&FP destination register, in the "Rd" field.

$<n>$ Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

$<m>$ Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>$Q$</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>$sz$</th>
<th>$Q$</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Vm>$ Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(eseize) element1;
bits(eseize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 &
IsMerging(fpcr);
bits(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
    end case

V[d] = result;
FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision (Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 0 | 1 1 1 1 0 0 | 0 1 1 0 0 1 0 | Rn | Rd

U op

FCMGE <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Scalar single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 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 0 0 1 0 | Rn | Rd

U op

FCMGE <V<d>, <V<n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
Vector half precision
(Armv8.2)

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

FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler Symbols

<Hd> is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
<d> is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\texttt{Vn}> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
b = bits(datasize) operand = \texttt{V}[n];
b = bits(datasize) result;
b = bits(esize) zero = FPZero('0');
b = bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = \texttt{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[]);
            \texttt{Elem}[result, e, esize] = if test_passed then Ones() else Zeros();
\texttt{V}[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMGT (register)**

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

### Scalar half precision

**(Armv8.2)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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></td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FCMGT <Hd>, <Hn>, <Hm>

if !**HaveFP16Ext**() then UNDEFINED;

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);
integer m = **UInt**(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
**CompareOp** cmp;
booleam abs;

case E:U:ac of
  when '000' cmp = **CompareOp_EQ**; abs = FALSE;
  when '010' cmp = **CompareOp_GE**; abs = FALSE;
  when '011' cmp = **CompareOp_GE**; abs = TRUE;
  when '110' cmp = **CompareOp_GT**; abs = FALSE;
  when '111' cmp = **CompareOp_GT**; abs = TRUE;
  otherwise UNDEFINED;

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</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></td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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;
else UNDEFINED;

Vector half precision
(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

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 '0000' 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;
else UNDEFINED;

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>
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**

<table>
<thead>
<tr>
<th>&lt;Hd&gt;</th>
<th>Is the 16-bit name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Hn&gt;</td>
<td>Is the 16-bit name of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Hm&gt;</td>
<td>Is the 16-bit name of the second SIMD&amp;FP source register, encoded in the &quot;Rm&quot; field.</td>
</tr>
<tr>
<td>&lt;V&gt;</td>
<td>Is a width specifier, encoded in &quot;sz&quot;:</td>
</tr>
<tr>
<td></td>
<td>sz</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>&lt;d&gt;</td>
<td>Is the number of the SIMD&amp;FP destination register, in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;n&gt;</td>
<td>Is the number of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;m&gt;</td>
<td>Is the number of the second SIMD&amp;FP source register, encoded in the &quot;Rm&quot; field.</td>
</tr>
<tr>
<td>&lt;Vd&gt;</td>
<td>Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;T&gt;</td>
<td>For the half-precision variant: is an arrangement specifier, encoded in &quot;Q&quot;:</td>
</tr>
<tr>
<td></td>
<td>Q</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>For the single-precision and double-precision variant: is an arrangement specifier, encoded in &quot;sz:Q&quot;:</td>
<td></td>
</tr>
<tr>
<td>sz</td>
<td>Q</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>&lt;Vn&gt;</td>
<td>Is the name of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Vm&gt;</td>
<td>Is the name of the second SIMD&amp;FP source register, encoded in the &quot;Rm&quot; field.</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(etype) element1;
bits(etype) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMGT (zero)

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(Armv8.2)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|   0      |   1      |   1      |   1      |   1      |   1      |   1      |   1      |   0      |   0      |   1      |   1      |   0      |   1      |   1      |   0      |   Rn     |   Rd     |
| U        | op       |
```

FCMGT <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Scalar single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 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      |   0      |   0      |   0      |   0      |   0      |   0      |   1      |   1      |   0      |   0      |   1      |   0      |   1      |   0      |   Rd     |
| U        | op       |
```

FCMGT <V<d>, <V<n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 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 |
|-----------------|-----------------|
| U               | op              |
```

FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

`CompareOp` comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 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               | op              |
```

FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

`CompareOp` comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler Symbols

- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<V>`: Is a width specifier, encoded in “sz”:

```
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>`: Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the half-precision variant: is an arrangement specifier, encoded in “Q”:
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | size| L  | M  | Rm | 0  | rot| 1  | H  | 0  | Rn | Rd |

(size == 01)

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

(size == 10)

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”: 
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<\(V_n\)> 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 "M:Rm" fields.

<\(T_s\)> Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T_s&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<\(T_i\)> Is the element index, encoded in "size:H:L":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T_i&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>

<\(R_o\)> Is the rotation, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>(&lt;R_o&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

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
  case rot of
    when '00'
      element1 = Elem[operand2, index*2, esize];
      element2 = Elem[operand1, e*2, esize];
      element3 = Elem[operand2, index*2+1, esize];
      element4 = Elem[operand1, e*2, esize];
    when '01'
      element1 = FPNeg(Elem[operand2, index*2+1, esize]);
      element2 = Elem[operand1, e*2+1, esize];
      element3 = Elem[operand2, index*2, esize];
      element4 = Elem[operand1, e*2+1, esize];
    when '10'
      element1 = FPNeg(Elem[operand2, index*2+1, esize]);
      element2 = Elem[operand1, e*2, esize];
      element3 = FPNeg(Elem[operand2, index*2, esize]);
      element4 = Elem[operand1, e*2+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+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;
Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | size | 0  | Rm | 1  | 1  | 0  | rot | 1  | Rn |  |  |  |  |  |  |  |  |  |  |  | 3 | 2 | 1 | 0 |

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":

[Note: The table and code snippet are placeholders and need to be replaced with the actual content from the document.]
<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>180</td>
</tr>
<tr>
<td>11</td>
<td>270</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand2, e*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, e*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, e*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize]);
            element4 = Elem[operand1, e*2+1, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 0 0 0 1 1 0 1 1 0          Rn           Rd
    U                     op
```

FCMLE <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  0 1 1 1 1 1 1 0 1 1 1 1 0 0 0 0 1 1 0 1 1 0          Rn           Rd
    U                     op
```

FCMLE <V<d>, <V<n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```
Vector half precision
(Armv8.2)

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

if !HaveFP16Ext() then UNDEFINED;

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
```

```plaintext
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

```plaintext
CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
```

```plaintext
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;
```

```plaintext
CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

- `<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":
  ```plaintext
  | sz | <V> |
  |------|
  | 0   | S   |
  | 1   | D   |
  ```
- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in "Q":

FCMLE (zero)
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bzelen = bits(datasize) operand = \text{V}[n];
bzelen = bits(datasize) result;
bzelen = bits(esize) zero = FPZero('0');
bzelen = bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when \text{CompareOp_GT} test_passed = FPCompareGT(element, zero, FPCR[]);
        when \text{CompareOp_GE} test_passed = FPCompareGE(element, zero, FPCR[]);
        when \text{CompareOp_EQ} test_passed = FPCompareEQ(element, zero, FPCR[]);
        when \text{CompareOp_LE} test_passed = FPCompareGE(zero, element, FPCR[]);
        when \text{CompareOp_LT} test_passed = FPCompareGT(zero, element, FPCR[]);
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

31 30 29 28 27 26 25 24 23 22 21 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 1 1 1 1 1 0 0 0 1 1 1 0 1 0 | Rn | Rd

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

31 30 29 28 27 26 25 24 23 22 21 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 1 0 0 0 0 0 1 1 1 0 1 0 | Rn | Rd

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)

31 30 29 28 27 26 25 24 23 22 21 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 | Rn | Rd

FCMLT (zero)
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

Vector single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 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 | sz | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | Rn | Rd |

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 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();`

```c
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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. This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in 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 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'

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;

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.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = if cmp_with_zero then FPZero('0') else V[m];
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.
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.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | ftype | 1   | Rm | 0  | 0  | 1  | 0  | 0  | 1  | x  | 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

```plaintext
integer n = UInt(Rn);
integer m = UInt(Rm); // ignored when opc<0> == '1'

integer datasize;

case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
data size = 16;
    else
      UNDEFINED;

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.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = if cmp_with_zero then FPZero('0') else V[m];
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.
FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the CPACR_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 1 1 1 1 0</td>
</tr>
</tbody>
</table>

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 = 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.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
result = if ConditionHolds(cond) then V[n] else V[m];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**FCVT**

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the `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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | 1  | opc | 1  | 0  | 0  | 0  | 0  | Rn  | Rd |

**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>`

```plaintext
type 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.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPAdvSIMDEnabled64();

bits(srcsize) operand = V[n];
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

Elem[result, 0, dstsize] = 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
FCVTAS <Vd>.<T>, <Vn>.<T>

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 \text{ DIV } esize;

\texttt{FPRounding} rounding = \texttt{FPRounding\_TIEAWAY};
boolean unsigned = (U == '1');

\textbf{Vector single-precision and double-precision}

\begin{center}
\begin{tabular}{cccccccccccc}
0 & | & Q & | & 0 & | & 0 & | & 1 & | & 1 & | & 0 & | & 0 & | & sz & | & 1 & | & 0 & | & 0 & | & 0 & | & 1 & | & 1 & | & 1 & | & 0 & | & 0 & | & 1 & | & 0 & | & Rn & | & Rd & | & U
\end{tabular}
\end{center}

FCVTAS <Vd>.<T>, <Vn>.<T>

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << \texttt{UInt}(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize \text{ DIV } esize;

\texttt{FPRounding} rounding = \texttt{FPRounding\_TIEAWAY};
boolean unsigned = (U == '1');

\textbf{Assembler Symbols}

\texttt{<Hd>} \quad Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Hn>} \quad Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\texttt{<V>} \quad Is a width specifier, encoded in "sz":

\begin{center}
\begin{tabular}{c|c}
sz & V
\hline
0 & S
1 & D
\end{tabular}
\end{center}

\texttt{<d>} \quad Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<n>} \quad Is the number of the SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Vd>} \quad Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<T>} \quad For the half-precision variant: is an arrangement specifier, encoded in "Q":

\begin{center}
\begin{tabular}{c|c}
Q & T
\hline
0 & 4H
1 & 8H
\end{tabular}
\end{center}

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\begin{center}
\begin{tabular}{c|c|c}
sz & Q & T
\hline
0 & 0 & 2S
0 & 1 & 4S
1 & 0 & \text{RESERVED}
1 & 1 & 2D
\end{tabular}
\end{center}

\texttt{<Vn>} \quad Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits( esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

### Half-precision to 32-bit (sf == 0 & ftype == 11) (Armv8.2)

FCVTAS <Wd>, <Hn>

### Half-precision to 64-bit (sf == 1 & ftype == 11) (Armv8.2)

FCVTAS <Xd>, <Hn>

### Single-precision to 32-bit (sf == 0 & ftype == 00)

FCVTAS <Wd>, <Sn>

### Single-precision to 64-bit (sf == 1 & ftype == 00)

FCVTAS <Xd>, <Sn>

### Double-precision to 32-bit (sf == 0 & ftype == 01)

FCVTAS <Wd>, <Dn>

### Double-precision to 64-bit (sf == 1 & ftype == 01)

FCVTAS <Xd>, <Dn>

```plaintext
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;
```

---

**Page 720**
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[],
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, FPRounding_TIEAWAY);
X[d] = intval;
```

Internal version only: isa v32.12, AdvSIMD v29.04, psudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTAU (vector)**

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 1 1 1 1 1 1 0 0 0 1 1 1 1 0 1 0 | Rn | Rd |

FCVTAU <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

### Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 0 | Rn | Rd |

FCVTAU <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

### Vector half precision

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 1 1 1 1 1 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 1 0 | Rn | Rd |

FCVTAU (vector)
FCVTAU <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>Q</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>sz</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
</table>

FCVTAU <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bias(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bias(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

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

CheckFPAvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, FPRounding_TIEAWAY);
X[d] = intval;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTL, FCVTL2**

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the FCVTL2 variant operates on the elements in the top 64 bits of the source register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
|0|Q|0|0|1|1|1|0|0|0|0|1|0|1|1|1|1|0| Rn | Rd  |

FCVTL{2} <Vd>, <Ta>, <Vn>,<Tb>
```

integer d = UInt(Rd);
ninteger n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

**Assembler Symbols**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></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 "sz":

<table>
<thead>
<tr>
<th></th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td></td>
</tr>
<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></th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR[]);
V[d] = result;
```

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  |     |
| U  | o2 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

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  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  |     |
| U  | o2 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0     |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|     |
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  |     |
| U  | o2 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
FCVTMS <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rn | Rd |
| U  | 02 | 01 |

FCVTMS <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```

---

Copyright © 2010-2020 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.
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

\texttt{CheckFPAdvSIMDEnabled64();}

\texttt{FPCRType fpcr = FPCR[];}
\texttt{bits(fltsize) fltval;}
\texttt{bits(intsize) intval;}

\texttt{fltval = V[n];}
\texttt{intval = FP\texttt{ToFixed}(fltval, 0, FALSE, fpcr, rounding);}
\texttt{X[d] = intval;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
## FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 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 |
| U | o2 | o1 |

FCVTMU <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 | 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 |
| U | o2 | o1 |

FCVTMU <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 | 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 |
| U | o2 | o1 |
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

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</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>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

bits(esize) element;
FPCRTyp e fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = E lems[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

V[d] = result;
```
FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
+-----------------+-----------------+
| 31  30  29  28  27  26  25  24  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  0  0  1 | 0  0  0  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 = FPDencodeRounding(rmode);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRType  fpcr = FPCR[];
bias(fltsize)  fltval;
bias(intsize)  intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding);
X[d] = intval;
FCVTN, FCVTN2

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

The FCVTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 0 0 1 1 1 0 0 | sz 1 0 0 0 0 1 0 1 1 0 1 0 | Rn | Rd
```

FCVTN(2) <Vd>,<Tb>, <Vn>..<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

- **2** Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

- **<Tb>** Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the “Rn” field.

- **<Ta>** Is an arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR[]);

Vpart[d, part] = result;
```
FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

```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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  |

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |

FCVTNS (vector)
```
FCVTNS $<Vd>$, $<Vn>$

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Rd | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

FCVTNS $<Vd>$, $<Vn>$

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

$<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Hn>$ Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

$<V>$ Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>$&lt;V&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

$<d>$ Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

$<n>$ Is the number of the SIMD&FP source register, encoded in the "Rn" field.

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTypfpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
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.
Half-precision to 32-bit ($\text{sf} == 0 \&\& \text{ftype} == 11$)  
(Armv8.2)

```
FCVTNS <Wd>, <Hn>
```

Half-precision to 64-bit ($\text{sf} == 1 \&\& \text{ftype} == 11$)  
(Armv8.2)

```
FCVTNS <Xd>, <Hn>
```

Single-precision to 32-bit ($\text{sf} == 0 \&\& \text{ftype} == 00$)

```
FCVTNS <Wd>, <Sn>
```

Single-precision to 64-bit ($\text{sf} == 1 \&\& \text{ftype} == 00$)

```
FCVTNS <Xd>, <Sn>
```

Double-precision to 32-bit ($\text{sf} == 0 \&\& \text{ftype} == 01$)

```
FCVTNS <Wd>, <Dn>
```

Double-precision to 64-bit ($\text{sf} == 1 \&\& \text{ftype} == 01$)

```
FCVTNS <Xd>, <Dn>
```

Integer $d = \text{UInt}(\text{Rd})$;
Integer $n = \text{UInt}(\text{Rn})$;

Integer intsize = if $\text{sf} == '1'$ 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
CheckFPAdvSIMEnabled64();

FPCRTYPE fpcr = FPCR[];
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

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 | 0 sz 1 0 0 0 0 | 1 1 0 1 0 1 0 | Rn | Rd
U o2 o1

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 1 0 0 | 1 1 0 1 0 1 0 | Rn | Rd
U o2 o1
FCVTNU <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**FPRounding** rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

### Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

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 = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

### Assembler Symbols

**<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Hn>** Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**<V>** Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

**<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<n>** Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Half-precision to 32-bit (sf == 0 && ftype == 11)  
(Armv8.2)

FCVTNU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(Armv8.2)

FCVTNU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTNU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTNU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTNU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTNU <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

FPRounding rounding;

case ftype of
when '00'
  fltsize = 32;
when '01'
  fltsize = 64;
when '10'
  UNDEFINED;
when '11'
  if HaveFP16Ext() then
    fltsize = 16;
  else
    UNDEFINED;

rounding = FPDencodeRounding(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.
CheckFPAdvSIMDEnabled64();

FPCRTypedef pcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPtoFixed(fltval, 0, TRUE, pcr, rounding);
X[d] = intval;
**FCVTPS (vector)**

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision (Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |
| U  | o2 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

```
FCVTPS <Hd>, <Hn>
if !HaveFP16Ext() then UNDEFINED;
    integer d = UInt(Rd);
    integer n = UInt(Rn);
    integer esize = 16;
    integer datasize = esize;
    integer elements = 1;
    FPRounding rounding = FPDecodeRounding(o1:o2);
    boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>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>1</td>
<td>sz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>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>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>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>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
FCVTPS <V><d>, <V><n>
    integer d = UInt(Rd);
    integer n = UInt(Rn);
    integer esize = 32 << UInt(sz);
    integer datasize = esize;
    integer elements = 1;
    FPRounding rounding = FPDecodeRounding(o1:o2);
    boolean unsigned = (U == '1');
```

### Vector half precision (Armv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

```
FCVTPS (vector)
FCVTPS <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------------------------------------|
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
| U | o2 | 1 | 0 |

FCVTPS <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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.

| sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
| sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd | rmode | opcode |

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
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)

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

CheckFPAdvSIMEnabled64();

FPCRTypelfcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFFixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;
FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in 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 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
```

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 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | Rd |
```

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)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rd |
```

FCVTPU (vector)
FCVTPU <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |
| U  | o2 | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  |

FCVTPU <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTyp fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bv128 result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in 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>0</th>
<th>ftype</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>round</td>
<td></td>
<td></td>
<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>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>
</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
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 fsize;
FPRounding  rounding;

case ftype of
    when '00'
        fsize = 32;
    when '01'
        fsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fsize = 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

CheckFPAdvSIMEnabled64();

FPCRTypelfpcr = FPCR[];
bitsufltsize) fltval;
bitsuinntsiz) intval;

fltval = V[n];
intval = FPToFricd(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**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | 1  | 1  | 0  | 1  | 0  | Rn  | Rd  |
```

FCVTXN <Vb><d>, <Va><n>

```plaintext
integer d = UInt(Rd);
ingert n = UInt(Rn);
if sz == '0' then UNDEFINED;
ingert esize = 32;
ingenert datasize = esize;
ingenert elements = 1;
ingenert part = 0;
```

**Vector**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | Rn  | Rd  |
```

FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>

```plaintext
integer d = UInt(Rd);
ingert n = UInt(Rn);
if sz == '0' then UNDEFINED;
genert esize = 32;
genert datasize = 64;
genert elements = 2;
genert part = UInt(Q);
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</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.

<Vb> 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>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
```

Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

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>

Is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

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>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(2*datasize) operand = V[n];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 &amp; IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], fpcr, FPRounding_ODD);

if merge then
    V[d] = result;
else
    Vpart[d, part] = Elem[result, 0, datasize];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each
element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and
writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in 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 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | != 0000 | immh | 1 | 1 | 1 | 1 | 1 | Rn | Rd |

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 data size = esize;
integer elements = 1;
integer frac bits = (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 | 1 | 1 | 1 | 1 | 1 | 0 | != 0000 | immh | 1 | 1 | 1 | 1 | 1 | Rn | Rd |

FCVTZS <Vd>..<T>, <Vn>..<T>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer data size = if Q == '1' then 128 else 64;
integer elements = data size DIV esize;

integer frac bits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

**Assembler Symbols**

<V> Is a width specifier, encoded in “immh”:

---

Page 769
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>  
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  
Is an arrangement specifier, encoded in "immh:Q":

<fbits>  
For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, frachbits, unsigned, fpcr, rounding);
V[d] = result;
```
FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|   |
| U |0 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | Rn |    |    |    |    |    |    |    |    | Rd |
| o2|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| o1|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

FCVTZS <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|   |
| U |0 1 | 1 | 1 | 1 | 1 | 0 | 1 | sz| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | Rn |    |    |    |    |    |    |    |    | Rd |
| o2|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| o1|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

FCVTZS <V>d>, <V>n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision
(Armv8.2)

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|   |
| U |0 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | Rn |    |    |    |    |    |    |    |    | Rd |
| o2|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| o1|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```
FCVTZS <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  | 0  | 1  | 1  | 1  | 0  | 1  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rn | Rd |
| U  | o2 | o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FCVTZS <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

**Assembler Symbols**

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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.

```plaintext
<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opcode</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
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11' if HaveFP16Ext() then fltsize = 16;
      else UNDEFINED;
if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
```
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".
For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, frachits, FALSE, fpcr, FPRounding_ZERO);
X[d] = intval;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn | Rd |
```

mode opcode
Half-precision to 32-bit (sf == 0 && ftype == 11)  
(ARMv8.2)

    FCVTZS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(ARMv8.2)

    FCVTZS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)

    FCVTZS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)

    FCVTZS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)

    FCVTZS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)

    FCVTZS <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
rounding = FPDecodeRounding(rmode);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypen fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = Y[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;
FCVTZU (vector, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 1 1 1 1 1 1 1 0 | != 0000          | immh             | 1 1 1 1 1 1 1    | Rn               | Rd               |
| U                |                  | immh             |

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 dat ssize = 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 1 0 1 1 1 1 1 0 | != 0000          | immh             | 1 1 1 1 1 1 1    | Rn               | Rd               |
| U                |                  | immh             |

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 dat ssize = if Q == '1' then 128 else 64;
integer elements = dat ssize 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":

FCVTZU (vector, fixed-point)
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>0H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</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>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding);
V[d] = result;
FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|
| 0 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| U | o2 | o1 |

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 |
|-------------------|-------------------|-------------------|
| 0 1 | 1 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| U | o2 | o1 |

FCVTZU <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 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 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| U | o2 | o1 |
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | Rn | Rd |
| o1 |   | o2|

FCVTZU <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == ‘10’ then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

bits(esimal) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

Half-precision to 32-bit (sf == 0 && ftype == 11)  
(Armv8.2)

FCVTZU <Wd>, <Hn>, #<fbits>

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(Armv8.2)

FCVTZU <Xd>, <Hn>, #<fbits>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTZU <Wd>, <Sn>, #<fbits>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTZU <Xd>, <Sn>, #<fbits>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTZU <Wd>, <Dn>, #<fbits>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTZU <Xd>, <Dn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".
For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```c
CheckFPAdvSIMDEnabled64();

FPCRTYPE fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, frachbits, TRUE, fpcr, FPRounding_ZERO);
X[d] = intval;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
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 = FPDencodeRounding(rmode);

Assembler Symbols

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = Y[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**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   |   |   | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | | 0 | 0 | 1 | 1 | 1 | 1 | | | | | | | | | | | | | | | | | | | | |
```

```
FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

### Single-precision and double-precision

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   |   |   | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | | | | | | | | | | | |
```

```
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>

**<Vn>**

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FDIV (scalar)**

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in **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 | Rm  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 0 0 | 0 0 | 0 1 | 1 1 | 1 0 |

**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>

```plaintext
integer d = UINT(Rd);
integer n = UINT(Rn);
integer m = UINT(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveF16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

**Assembler Symbols**

- `<Dd>`   Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>`   Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>`   Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>`   Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`   Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>`   Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>`   Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>`   Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>`   Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypelfcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPDiv(operand1, operand2, FPCR[]);

V[d] = result;
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 represented as a signed 32-bit integer, then the result is the integer modulo $2^{32}$, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Double-precision to 32-bit**

*(Armv8.3)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn |
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn |

sf

ftype

rmode

opcode

FJCVTZS <Wd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

if !HaveFJCVTZSExt() then UNDEFINED;

**Assemble Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];

bits(64) fltval;

bits(32) intval;

bit Z;

fltval = V[n];

(intval, Z) = FPToFixedJS(fltval, fpcr, TRUE);

PSTATE.<N,Z,C,V> = '0':Z:'00';

X[d] = intval;

**FMADD**

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision (ftype == 11)

**(Armv8.2)**

FMADD <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (ftype == 00)

FMADD <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (ftype == 01)

FMADD <Dd>, <Dn>, <Dm>, <Da>

```plaintext
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
end case;
```

**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

```plaintext
CheckFPAdvSIMDEnabled64();

bits(esize) operanda = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d] = result;
```

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |
| U | o1 | |

FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 1 | 1 | 0 | 1 | Rn | Rd |
| U | o1 | |

FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

| 0 | 4H |
| 1 | 8H |

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  
  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in $FPSCR$, 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.

```
0 0 0 1 1 1 1 0 | ftype | 1 | Rm | 0 1 0 0 1 0 | Rn | Rd
```

Half-precision ($ftype == 11$)
(Armv8.2)

FMAX $<Hd>$, $<Hn>$, $<Hm>$

Single-precision ($ftype == 00$)

FMAX $<Sd>$, $<Sn>$, $<Sm>$

Double-precision ($ftype == 01$)

FMAX $<Dd>$, $<Dn>$, $<Dm>$

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

$<Dd>$ Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
$<Dn>$ Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
$<Dm>$ Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
$<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
$<Hn>$ Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
$<Hm>$ Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
$<Sd>$ Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
$<Sn>$ Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
$<Sm>$ Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

`CheckFPAdvSIMDEnabled64();`
`bits(eseize) operand1 = V[n];`
`bits(eseize) operand2 = V[m];`

`FPCRTypedefpcr = FPCR[];`
`boolean merge = IsMerging(fpcr);`
`bits(128) result = if merge then V[n] else Zeros();`

`Elem[result, 0, esize] = FPMAX(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 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
| U | a |

FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|---------------------------|-------------------|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| U | o1 |

FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 0 ftype 1 Rm 0 1 1 0 1 0 Rn Rd 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>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
    case ftype of
        when '00' esize = 32;
        when '01' esize = 64;
        when '10' UNDEFINED;
        when '11'
            if HaveFP16Ext() then
                esize = 16;
            else
                UNDEFINED;
```

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypedef pcr = FPCR[];
boolean merge = IsMerging(pcr);
bites(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);
V[d] = result;
**FMAXNMP (scalar)**

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

*(Armv8.2)*

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 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   | 1   | 1   | 0   | 0   | 1   | 0   | Rn  | Rd  |    |    |    |    |    |    |    |    |    |

**FMAXNMP <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   | sz  | 1   | 1   | 0   | 0   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | Rn  | Rd  |    |    |    |    |    |    |    |    |    |

**FMAXNMP <V><d>, <Vn>.<T>**

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;

### Assembler Symbols

**<V>**

For the half-precision variant: is the destination width specifier, encoded in "sz":

```
sz <V>
0   H
1   RESERVED
```

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

```
sz <V>
0   S
1   D
```

**<d>**

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>**

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<T>**

For the half-precision variant: is the source arrangement specifier, encoded in "sz":

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":
For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 | 0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
| U | a |

FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

to integer d = UInt(Rd);
to integer n = UInt(Rn);
to integer m = UInt(Rm);
to integer esize = 16;
to integer datasize = if Q == '1' then 128 else 64;
to 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 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| U | o1 |

FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
to integer n = UInt(Rn);
to integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
to integer esize = 32 << UInt(sz);
to integer datasize = if Q == '1' then 128 else 64;
to 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>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 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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 \textit{FMAX (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**

\textbf{(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 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |    Rd
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Rn
01

FMAXNMV <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
\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
0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |    Rd
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Rn
01

FMAXNMV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
\end{verbatim}

**Assembler Symbols**

\(<V>\) For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

\[
\begin{array}{c|c}
\text{sz} & <V> \\
\hline
0 & S \\
1 & RESERVED \\
\end{array}
\]

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Vn}>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<\text{T}>\) 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>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>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>01 sz</td>
</tr>
</tbody>
</table>

FMAXP <V><d>, <Vn><T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;

Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 0</td>
</tr>
<tr>
<td>01 sz</td>
</tr>
</tbody>
</table>

FMAXP <V><d>, <Vn><T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values. This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR** or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

**Half-precision** *(Armv8.2)*

<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>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>1</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>
</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>

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**

<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>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>1</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>
</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>

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>

For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vn>**  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>**  
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem(concat, 2*e, esize);
    element2 = Elem(concat, (2*e)+1, esize);
  else
    element1 = Elem(operand1, e, esize);
    element2 = Elem(operand2, e, esize);
  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | | Rn | Rd |

FMAXV <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | | Rn | Rd |

FMAXV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.
For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

FMAXV
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMIN (vector)**

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

### Half-precision

*(Armv8.2)*

| 31 30 29 28 27 26 25 24 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               |
| 0               | 1               | 1               | 1               |
| 0               | 1               | 0               | 1               |
| Rm              | 0               | 0               | 1               |
| 0               | 1               | 1               | 0               |
| Rd              |                 |                 |                 |
|                 |                 |                 |                 |

**FMIN <Vd>..<T>, <Vn>..<T>, <Vm>..<T>**

```plaintext
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 |
|-----------------|-----------------|-----------------|-----------------|
| O               | Q               | 0               | 0               |
| 0               | 1               | 1               | 1               |
| 0               | 1               | 0               | 1               |
| 1               | sz             | 1               |                 |
| Rm              | 1               | 1               | 1               |
| 1               | 1               | 0               | 1               |
| Rd              |                 |                 |                 |
|                 |                 |                 |                 |

**FMIN <Vd>..<T>, <Vn>..<T>, <Vm>..<T>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the “Rd” field.
- **<T>** For the half-precision variant: is an arrangement specifier, encoded in “Q”:

```
<table>
<thead>
<tr>
<th>0</th>
<th>4H</th>
</tr>
</thead>
</table>
```

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

FMIN (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(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMIN (scalar)**

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ftype</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</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>
<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>
<td></td>
</tr>
</tbody>
</table>
```

**Half-precision (ftype == 11)**

(Armv8.2)

FMIN <Hd>, <Hn>, <Hm>

**Single-precision (ftype == 00)**

FMIN <Sd>, <Sn>, <Sm>

**Double-precision (ftype == 01)**

FMIN <Dd>, <Dn>, <Dm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypenpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem(result, 0, esize) = FPMin(operand1, operand2, fpcr);
V[d] = result;
**FMINNM (vector)**

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision (Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------------------|-------------------------------------------------|-------------------------------------------------|
| 0 | O | 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 | O | 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.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q".
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{Vm}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \text{V}[n];
bites(datasize) operand2 = \text{V}[m];
bites(datasize) result;
bites(2*datasize) concat = operand2:operand1;
bites(esize) element1;
bites(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = \text{Elem}[concat, 2*e, esize];
        element2 = \text{Elem}[concat, (2*e)+1, esize];
    else
        element1 = \text{Elem}[operand1, e, esize];
        element2 = \text{Elem}[operand2, e, esize];
    if minimum then
        \text{Elem}[result, e, esize] = FPMinNum(element1, element2, FPCR[]);  
    else
        \text{Elem}[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
\text{V}[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNM (scalar)

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, 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 1 1 0 Rn Rd

op

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 esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bites(esize) operand2 = V[m];

FPCRTyppe fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bites(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = 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  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |

FMINNMP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;

### Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | sz | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rd | Rn |

FMINNMP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;

### Assembler Symbols

- `<V>` For the half-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<T>` For the half-precision variant: is the source arrangement specifier, encoded in "sz":


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();
bvts(bits(datasize)) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 1 | 1 | 1 | 1 | 0 | O | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 1 | Rn | Rd | U

FMINNMP <Vd>..<T>, <Vn>..<T>, <Vm>..<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

Single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 1 | 1 | 1 | 0 | 1 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 1 | Rn | Rd | U

FMINNMP <Vd>..<T>, <Vn>..<T>, <Vm>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>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 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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNMV

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 FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(Armv8.2)

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 1   | 1   | 1   | 0   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | Rn  |   Rd |

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;

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   | sz  | 1   | 1   | 0   | 0   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | Rn  |   Rd |

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;

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":

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINS (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Artmv8.2)

| 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  | 1  | 1  | 0  |   |   |   |   |   |

FMINS <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  | 1  | 1  | 1  | 1  | 0  |   |   |   |   |   |

FMINS <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;

**Assembler Symbols**

<V> For the half-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMIN, operand, esize);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 1  | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |
| o1 |     |

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 1  | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | Rm | 1 | 1 | 1 | 1 | 0 | 1 | Rn | Rd |
| o1 |     |

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":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

Page 832
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rd |

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

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rd |

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.

<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_FMIN, operand, esize);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision
(Armv8.2)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|----------------|
| Rd              | Rn              | M              | L              |

0 1 0 1 1 1 1 1 0 0 L M Rm 0 0 0 1 H 0 Rn 0 0 0 1 H 0 Rd
```

FMLA <Hd>, <Hn>, <Vm>.H[<index>]

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Scalar, single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|----------------|
| Rd              | Rn              | M              | L              |

0 1 0 1 1 1 1 1 1 1 1 1 sz L M Rm 0 0 0 1 H 0 Rn 0 0 0 1 H 0 Rd
```

FMLA <V>d>, <V>n>, <Vm>.<Ts>[<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);

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 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | L | M | Rm | 0 | 0 | 0 | 1 | H | 0 | Rn | Rd |

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 | 0 | 0 | 1 | 1 | 1 | 1 | sz | L | M | Rm | 0 | 0 | 0 | 1 | H | 0 | Rn | Rd |

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 half-precision variant: is an arrangement specifier, encoded in "Q";
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
<td></td>
</tr>
</tbody>
</table>

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.

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>

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, in the range V0 to V15, encoded in the "Rm" field.

Is an element index, in the range 0 to 7, encoded in the "H:L:M" fields.

Is the element index, encoded in "sz:L:H":

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(idxsiz3size) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRTyps fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

```
 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
    |   | Rm|     |     |     |     | 0 | 0 | 0 | 1 | 1 |
```

\[a\]

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**

```
 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 |
    |   | Rm|     |     |     |     | 1 | 1 | 0 | 1 | 1 |
```

\[op\]

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":

```
 Q  <T>
 0  4H
 1  8H
```

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":


<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLAL, FMLAL2 (by element)

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_AA64ISAR0_EL1. FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL

(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 1 1 1 0 | L M | Rm 0 0 0 0 | H 0 | Rn | Rd

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
0 1 0 1 1 1 1 1 0 | L M | Rm 1 0 0 0 | H 0 | Rn | Rd

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 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>0</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>0</th>
<th>2H</th>
</tr>
</thead>
<tbody>
<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

```c
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLAL, FMLAL2 (vector)

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL

(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 <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 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in "Q":
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vb> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>0</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAvSIMDEnabled64();
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

FMLS <Hd>, <Hn>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

Scalar, single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 1 1 | sz | L M | Rm 0 1 0 1 H 0 | Rn Rd

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)

```
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

```
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;

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":
  - `sz <V>
    - 0: S
    - 1: D
- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in "Q":
  - `FMLS (by element)`
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
<th>sz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
<td>4S</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: 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>

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, 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.

Is an element size specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

Is the element index, encoded in "sz:L:H".

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(idxsiz) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRTyp e fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);
V[d] = result;
```
FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

```
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

```
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”:

```
<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 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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in $FPCR$, the exception results in either a flag being set in $FPSR$, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID $AA64ISAR0_EL1$. FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL
(Armv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | L  | M  | Rm | 0  | 1  | 0  | 0  | H  | 0  | Rn | 0  | Rd |
| sz | S  |

FMLSL $<Vd>$. $<Ta>$, $<Vn>$. $<Tb>$, $<Vm>.H[<index>]$

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2
(Armv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | L  | M  | Rm | 1  | 1  | 0  | 0  | H  | 0  | Rn | 0  | Rd |
| sz | S  |

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 <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 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:
<Ta> 0 2S
1 4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

<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**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the \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{Half-precision} and \texttt{Single-precision and double-precision}

### 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 0 1 1 1 1 0 0 0 0 0 0 a b c 1 1 1 1 1 1 d e f g h</td>
</tr>
</tbody>
</table>

FMOV <Vd>.<T>, #<imm>

\begin{verbatim}
if !HaveFP16Ext() then UNDEFINED;
integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);
imm = Replicate(imm16, datasize DIV 16);
\end{verbatim}

### 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 0 1 1 1 1 0 0 0 0 0 0 a b c 1 1 1 1 1 0 1 d e f g h</td>
</tr>
</tbody>
</table>

#### Single-precision (op == 0)

FMOV <Vd>.<T>, #<imm>

#### Double-precision (Q == 1 && op == 1)

FMOV <Vd>.2D, #<imm>

\begin{verbatim}
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 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
\end{verbatim}

Assembler Symbols

\begin{itemize}
  \item \texttt{<Vd>} Is the name of the SIMD\&FP destination register, encoded in the "Rd" field.
  \item \texttt{<T>} For the half-precision variant: is an arrangement specifier, encoded in "Q":
\end{itemize}
For the single-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions.

Operation

```c
CheckFPAdvSIMDEnabled64();

V[rd] = imm;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | ftype | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

opc

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 esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
end

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the 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(esize) operand = V[n];

Elem[Zeros()], 0, esize = operand;
V[d] = Zeros();
**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.

| sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | x | 1 | 1 | x | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
|----|---|---|---|---|---|---|---|-------|---|---|---|---|---|---|---|---|---|---|---|---|
|    |   |   |   |   |   |   |   | rmode |   |   |   |   |   |   |   |   |   |   |   |   |   |

```
Half-precision to 32-bit (sf == 0 & ftype == 11 & rmode == 00 & opcode == 110)
(Armv8.2)
FMOV <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 & ftype == 11 & rmode == 00 & opcode == 110)
(Armv8.2)
FMOV <Xd>, <Hn>

32-bit to half-precision (sf == 0 & ftype == 11 & rmode == 00 & opcode == 111)
(Armv8.2)
FMOV <Hd>, <Wn>

32-bit to single-precision (sf == 0 & ftype == 00 & rmode == 00 & opcode == 111)
FMOV <Sd>, <Wn>

Single-precision to 32-bit (sf == 0 & ftype == 00 & rmode == 00 & opcode == 110)
FMOV <Wd>, <Sn>

64-bit to half-precision (sf == 1 & ftype == 11 & rmode == 00 & opcode == 111)
(Armv8.2)
FMOV <Hd>, <Xn>

64-bit to double-precision (sf == 1 & ftype == 01 & rmode == 00 & opcode == 111)
FMOV <Dd>, <Xn>

64-bit to top half of 128-bit (sf == 1 & ftype == 10 & rmode == 01 & opcode == 111)
FMOV <Vd>.D[1], <Xn>

Double-precision to 64-bit (sf == 1 & ftype == 01 & rmode == 00 & opcode == 110)
FMOV <Xd>, <Dn>

Top half of 128-bit to 64-bit (sf == 1 & ftype == 10 & rmode == 01 & opcode == 110)
FMOV <Xd>, <Vn>.D[1]
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx'    // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00'    // [US]CVTF
    rounding = FPRoundingMode(FPCR[]);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00'    // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00'    // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01'    // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64;  // size of D[1] is 64
  when '11 11'    // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```java
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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
    fltval = X[n];
    intval = intval<fsize-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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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, 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 1 | ftype | imm8 | 1 0 0 0 0 0 0 | Rd
```

Half-precision (ftype == 11)
(Armv8.2)

FMOV <Hd>, #<imm>

Single-precision (ftype == 00)

FMOV <Sd>, #<imm>

Double-precision (ftype == 01)

FMOV <Dd>, #<imm>

```
integer d = UInt(Rd);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11' if HaveFP16Ext() then
data size = 16;
else
  UNDEFINED;

bits(datasize) imm = VFPExpandImm(imm8);
```

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions.

Operation

```
CheckFPAdvSIMDEnabled64();

V[d] = imm;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 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 ftype 0 | Rm | 1 | Ra | Rn | Rd

01 00

Half-precision (ftype == 11)

(Armv8.2)

FMSUB <Hd>, <Hn>, <Hm>, <Ha>

Single-precision (ftype == 00)

FMSUB <Sd>, <Sn>, <Sm>, <Sa>

Double-precision (ftype == 01)

FMSUB <Dd>, <Dn>, <Dm>, <Da>

integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMUL (by element)

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in 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 0 0 L M</td>
</tr>
</tbody>
</table>

FMUL <Hd>, <Hn>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

Scalar, single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

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  | 0  | 0  | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

U

FMUL <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 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  | sz | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

U

FMUL <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 half-precision variant: is an arrangement specifier, encoded in "Q":

FMUL (by element)
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsizex) operand2 = V[m];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);
V[d] = result;
```

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  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 0  | 1  | 1  | 1  | Rn | | Rd |

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 1  | 1  | 1  | Rn | | Rd |

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMUL (scalar)**

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
    31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
     0  0  1  1  1  0  ftype  1  |  Rm  |  0  0  0  0  1  0 |  Rn  |  Rd
    op
```

**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
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
    case ftype of
        when '00' esize = 32;
        when '01' esize = 64;
        when '10' UNDEFINED;
        when '11'
            if HaveFP16Ext() then
                esize = 16;
            else
                UNDEFINED;
```

**Assembler Symbols**

- **<Dd>** is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Dn>** is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Dm>** is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Hd>** is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Hm>** is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Sd>** is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Sn>** is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Sm>** is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

bits(esize) product = FPMul(operand1, operand2, fpcr);
Elem[result, 0, esize] = product;
V[d] = result;
**FMULX (by element)**

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR** or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision.

### Scalar, half-precision (Armv8.2)

```plaintext
FMULX <Hd>, <Hn>, <Vm>.H<index>
```

```plaintext
if !HaveFP16Ext() then UNDEFINED;
```

```plaintext
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);
```

```plaintext
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

### Scalar, single-precision and double-precision

```plaintext
FMULX <V<d>, <V<n>, <Vm>.<Ts>[<index>]
```

```plaintext
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
```

```plaintext
case sz:L of
    when '0x' index = UINT(H:L);
    when '10' index = UINT(H);
    when '11' UNDEFINED;
```

```plaintext
integer d = UINT(Rd);
integer n = UINT(Rn);
integer m = UINT(Rmhi:Rm);
```

```plaintext
integer esize = 32 << UINT(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```
Vector, half-precision  
(Armv8.2)  

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 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  

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 half-precision variant: is an arrangement specifier, encoded in “Q”:
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4S</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":
| sz  | <Ts> |
| 0   | S    |
| 1   | D    |

| <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":
| sz | L | <index> |
| 0  | x | H:L     |
| 1  | 0 | H       |
| 1  | 1 | RESERVED |

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(idxsizsize) operand2 = V[m];
bits(esize) element1;
bits(esize) element2 = Elem(operand2, index, esize);
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && Merge(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
    element1 = Elem(operand1, e, esize);
    if mulx op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMULX**

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

**Scalar half precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | 0  | 1  | 1  | 1  |  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**FMULX** <Hd>, <Hn>, <Hm>

if !**HaveFP16Ext**() then UNDEFINED;

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);
integer m = **UInt**(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

**Scalar single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | 1  | 1  | 1  |  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**FMULX** <V>d>, <V>n>, <V>m>

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);
integer m = **UInt**(Rm);
integer esize = 32 **UInt**(sz);
integer datasize = esize;
integer elements = 1;

**Vector half precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | 0  | 1  | 1  | 1  |  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**FMULX** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !**HaveFP16Ext**() then UNDEFINED;

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);
integer m = **UInt**(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FMULX** <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz: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**

<Hp> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FMULX(element1, element2, fpcr);
V[d] = result;

FMULX
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)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

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

| 31 30 29 28 27 26 25 24 23 22 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 | sz | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

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":

<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];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register. Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Hexadecimal representation](image)

**Half-precision (ftype == 11)**
\textit{(Armv8.2)}

\texttt{FNEG <Hd>, <Hn>}

**Single-precision (ftype == 00)**

\texttt{FNEG <Sd>, <Sn>}

**Double-precision (ftype == 01)**

\texttt{FNEG <Dd>, <Dn>}

integer \(d = \text{UInt}(Rd);\)
integer \(n = \text{UInt}(Rn)\);

integer esize;
\textbf{case} ftype \textbf{of}
\textbf{when} '00' esize = 32;
\textbf{when} '01' esize = 64;
\textbf{when} '10' \text{UNDEFINED};
\textbf{when} '11'
\textbf{if} \texttt{HaveFP16Ext()} \textbf{then}
    esize = 16;
\textbf{else}
    \text{UNDEFINED};

**Assembler Symbols**

\texttt{<Dd>}  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Dn>}  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Hd>}  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Hn>}  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Sd>}  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Sn>}  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

\texttt{CheckFPAdvSIMDEnabled64();}

\texttt{FPCRType fpcr = FPCR[];}
\texttt{boolean merge = IsMerging(fpcr);}
\texttt{bits(128) result = if merge then \text{V}[d] else \text{Zeros}();}

\texttt{bits(esize) operand = \text{V}[n];}
\texttt{Elem[result, 0, esize] = FPNeg(operand);}
\texttt{\text{V}[d] = result;}

FNEG (scalar)
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | ftype | 1 | Rm | 0 | Ra | 0 | Rn | 0 | Rd | 0 |

### Half-precision (ftype == 11) (Armv8.2)

FNMADD <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (ftype == 00)

FNMADD <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (ftype == 01)

FNMADD <Dd>, <Dn>, <Dm>, <Da>

```plaintext
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
end case;
```

### Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Da>` Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Ha>` Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand = FPNeg(operand);
operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNMSUB

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.

```
0 0 0 1 1 1 1 1 ftype 1 Rm 1 Ra Rn Rd
```

**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 esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Da>` Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
Is the 16-bit name of the third SIMD&FP source register holding the minuend, 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 minuend, encoded in the "Ra" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(esize) operanda = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand = FPNeg(operand);
Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<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>ftype</td>
<td>1</td>
<td>Rm</td>
<td>1</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>
</tr>
</tbody>
</table>

op

Half-precision (ftype == 11)  
(Armv8.2)

FNMUL <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FNMUL <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FNMUL <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;

switch ftype 
    case '00': esize = 32;
    case '01': esize = 64;
    case '10': UNDEFINED;
    case '11':
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypenpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

bits(esize) product = FPMul(operand1, operand2, fpcr);
product = FPNeg(product);
Elem[result, 0, esize] = product;

V[d] = result;
FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 1 0

FRECPE <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 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 sz 1 0 0 0 0 1 1 1 0 1 1 0

FRECPE <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision

(Armv8.2)

31 30 29 28 27 26 25 24 23 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 1 0 0 1 1 1 0 1 1 0

FRECPE <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

FRECPE \(<V_d>.<T>, <V_n>.<T>\)

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

\(<H_d>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<H_n>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\(<V>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.
\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<V_n>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

FPCRTyp e fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR[]);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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   |     |   0   0   1   1   1   1   |                    |
|         Rm         |     |         Rn         |                    |
|         Rd         |

FRECPS <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

(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   | sz |     |   1   1   1   1   1   |                    |
|         Rm         |     |         Rn         |                    |
|         Rd         |

FRECPS <V>d>, <V>n>, <V>m

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 "UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 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   1   1   1   |                    |
|         Rm         |     |         Rn         |                    |
|         Rd         |

FRECPS <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

FRECPS $<Vd>.<T>$, $<Vn>.<T>$, $<Vm>.<T>$

integer $d =$ $\text{UInt}(Rd)$;
integer $n =$ $\text{UInt}(Rn)$;
integer $m =$ $\text{UInt}(Rm)$;
if $sz:Q == '10'$ then UNDEFINED;
integer $esize = 32 << \text{UInt}(sz)$;
integer $datasize = \text{if } Q == '1' \text{ then } 128 \text{ else } 64$;
integer $elements = datasize \text{ DIV } esize$;

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 half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>$Q$</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>$sz$</th>
<th>$Q$</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
$<Vm>$ Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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**, **CPR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

![Hexadecimal encoding](image)

**FRECPX** `<Hd>`, `<Hn>`

```java
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

![Hexadecimal encoding](image)

**FRECPX** `<V>`<d>, `<V>`<n>`

```java
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

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \text{\texttt{V}}[n];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && \text{\texttt{IsMerging}}(fpcr);
bits(128) result = if merge then \text{\texttt{V}}[d] else \text{\texttt{Zeros}}();
bits(esize) element;
for e = 0 to elements-1
  element = \text{\texttt{Elem}}(operand, e, esize);
  \text{\texttt{Elem}}(result, e, esize) = \text{\texttt{FPRecpX}}(element, fpcr);
\text{\texttt{V}}[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register. A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(Armv8.5)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>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>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>U</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
<T> Is an arrangement specifier, encoded in "sz: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>
<tr>
<td>0</td>
<td>RESERVED</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.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
   element = Elem[operand, e, esize];
   Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
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  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |

ftype op

If ftype == 00

FRINT32X <Sd>, <Sn>

If ftype == 01

FRINT32X <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;
FRounding 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

```c
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | Rn | Rd |

FRINT32Z <Vd>.<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = 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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | x  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

ftype  op  Rn  Rd

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 esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;

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();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 32);
V[d] = result;
**FRINT64X (vector)**

Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the **FPCR**, and writes the result to the SIMD&FP destination register. A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Vector single-precision and double-precision**
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | Rn | Rd |
| U  | op |

**FRINT64X** `<Vd><T>, <Vn><T>`

```plaintext
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**

```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] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
FRINT64X (scalar)

Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the **FPCR**, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Floating-point (Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | x  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 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 esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then \texttt{V}[d] else \texttt{Zeros}();
bits(esize) operand = \texttt{V}[n];

\texttt{Elem}[result, 0, esize] = \texttt{FPRoundIntN}(operand, fpcr, rounding, 64);

\texttt{V}[d] = result;
**FRINT64Z (vector)**

Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector single-precision and double-precision

(ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 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 "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
def 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 corresponding input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns 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  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  |
| ftype | op |

Single-precision (ftype == 00)

FRINT64Z <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT64Z <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bis(128) result = if merge then V[d] else Zeros();
bis(esize) operand = V[n];
Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 64);
V[d] = result;
**FRINTA (vector)**

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision
(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| O                | Q                | 1                |
| 0                | 1                | 0                |
| 1                | 1                | 1                |
| 1                | 1                | 0                |
| Rd               | Rn               |

\[
\text{FRINTA} \quad \langle \text{Vd}\rangle . \langle \text{T}\rangle , \quad \langle \text{Vn}\rangle . \langle \text{T}\rangle 
\]

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer dataset = if Q == '1' then 128 else 64;
integer elements = dataset DIV esize;

boolean exact = FALSE;

FPRounding rounding;
case U:0:1:0 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                | Q                | 1                |
| 0                | 1                | 0                |
| 1                | 1                | 1                |
| 1                | 0                | 0                |
| Rd               | Rn               |

\[
\text{FRINTA} \quad \langle \text{Vd}\rangle . \langle \text{T}\rangle , \quad \langle \text{Vn}\rangle . \langle \text{T}\rangle 
\]
FRINTA \(<Vd>\).\(<T>\), \(<Vn>\).\(<T>\)

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 = FPDetectRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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 \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.

### Half-precision (ftype == 11)

\texttt{FRINTA <Hd>, <Hn>}

### Single-precision (ftype == 00)

\texttt{FRINTA <Sd>, <Sn>}

### Double-precision (ftype == 01)

\texttt{FRINTA <Dd>, <Dn>}

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);

type esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

### Assembler Symbols

\texttt{<Dd>} is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Dn>} is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Hd>} is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Hn>} is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Sd>} is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Sn>} is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();

FPCRTypetype fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bm(128) result = if merge then V[d] else Zeros();
bm(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, FPRounding_TIEAWAY, FALSE);

V[d] = result;
\end{verbatim}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FRINTI (vector)**

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the **FPCR**, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

### Half-precision (Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
| \(U\) | 0 | 2 | \(o_1\) |

**FRINTI** \(<Vd>.<T>, <Vn>.<T>\)

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
n = UInt(Rn);

integer esize = 16;
integer datatype = if Q == '1' then 128 else 64;
integer elements = datatype DIV esize;

boolean exact = FALSE;
FProunding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FProunding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FProundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FProundingMode(FPCR[]);
```

### Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
| \(U\) | 0 | 2 | \(o_1\) |

FRINTI (vector)
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[]);
endcase;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;
FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 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 1 1 1 0 0 0 0 Rn Rd
```

rmode

**Half-precision (ftype == 11)**
(Armv8.2)

FRINTI <Hd>, <Hn>

**Single-precision (ftype == 00)**

FRINTI <Sd>, <Sn>

**Double-precision (ftype == 01)**

FRINTI <Dd>, <Dn>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
switch ftype {
    case '00' : esize = 32;
    case '01' : esize = 64;
    case '10' : UNDEFINED;
    case '11' : 
        if HaveFP16Ext() then 
            esize = 16;
        else 
            UNDEFINED;
    }

FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypen fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in $FPCR$, the exception results in either a flag being set in $FPSR$, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| U | 0 | 1 |

FRINTM <$Vd$.<T>, <$Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:01:02 of
  when '0xx' rounding = FPDecodeRounding(01:02);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

### Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| U | 0 | 1 |
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[]);

Assemble 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.

![Instruction Format](image)

<table>
<thead>
<tr>
<th>FINTM (scalar)</th>
<th>Page 920</th>
</tr>
</thead>
</table>

Half-precision (ftype == 11)
(Armv8.2)

FRINTM <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTM <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTM <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
FPRounding rounding;
rounding = FPDereoceRounding('10');

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypet fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
b bits(128) result = if merge then V[d] else Zeros();
b bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision
(Armv8.2)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| O | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |

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;

FP Rounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]): exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

### Single-precision and double-precision

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| O | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDdecodeRounding(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;
FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in $FPCR$, the exception results in either a flag being set in $FPSR$, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 1 0 | ftype | 1 0 0 1 0 0 0 1 1 0 0 0 0 | Rd
```

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 esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
FPRounding rounding;
rounding = FPDetectRounding('00');
```

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypew fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bital(128) result = if merge then \texttt{V[d]} else \texttt{Zeros}();
bital(esize) operand = \texttt{V[n]};

\texttt{Elem}\{result, 0, esize\} = \texttt{FPRoundInt}(operand, fpcr, rounding, false);

\texttt{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)**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

**FRINTP <Vd>.<T>, <Vn>.<T>**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;

**FPRounding** rounding;

case U:o1:o2 of

  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); 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>
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;
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":

\[
\begin{array}{c|c}
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\[
\begin{array}{c|c|c|c|c}
sz & Q & <T> \\
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & RESERVED \\
1 & 1 & 2D \\
\end{array}
\]

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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.

```

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |
```

### 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 esize;

```java
    case ftype of
        when '00' esize = 32;
        when '01' esize = 64;
        when '10' UNDEFINED;
        when '11'
            if HaveFP16Ext() then
                esize = 16;
            else
                UNDEFINED;
    FPRounding rounding;
    rounding = FPDDecodeRounding('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();

FPCRTypelfcr = FPCR[];
boolean merge = IsMerging(fpcr);
binalg(f128) result = if merge then V[d] else Zeros();
binalg(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the 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</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>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>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>

UR

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;

U:0:1:0

when '0xx' rounding = FPDecodeRounding(o1:o2);
when '100' rounding = FPRounding_TIEAWAY;
when '101' UNDEFINED;
when '110' rounding = FPRoundingMode(FPCR[]);
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>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</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>

UR

FRINTX (vector)
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 = FPParseRounding(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;
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.

Half-precision (ftype == 11)
(Armv8.2)

FRINTX <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTX <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTX <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
  end

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.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = 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 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 1 1 0 |
| U 02 o2 o1 |

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:01: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 1 1 1 0 1 1 0 0 0 0 1 1 0 0 1 1 0 |
| U 02 o2 o1 |
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
  when '0xx' \( \text{rounding} = \text{FPDecodeRounding}(o1:o2) \);
  when '100' \( \text{rounding} = \text{FPRounding}\_\text{TIEAWAY} \);
  when '101' \( \text{UNDEFINED} \);
  when '110' \( \text{rounding} = \text{FPRoundingMode}(\text{FPCR}[]) \); \( \text{exact} = \text{TRUE} \);
  when '111' \( \text{rounding} = \text{FPRoundingMode}(\text{FPCR}[]) \);

\textbf{Assembler Symbols}

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>( Q )</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>( sz )</th>
<th>( Q )</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\textbf{Operation}

\texttt{CheckFPAdvSIMDEnabled64()};
\texttt{bits(datasize) operand = \( V[n] \)};
\texttt{bits(datasize) result};
\texttt{bits(esize) element};

for \( e = 0 \) to \( \text{elements}-1 \)
  \texttt{element = \text{Elem}[operand, e, esize]};
  \texttt{Elem[result, e, esize] = \text{FPRoundInt}(element, FPCR[], \text{rounding}, \text{exact})};
\texttt{V[d] = 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 0 0 0 0 Rn Rd

rmode

Half-precision (ftype == 11)
(Armv8.2)

FRINTZ <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTZ <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTZ <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
  FP.Rounding rounding;
  rounding = FPD DecodeRounding('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();

FPCRTypetype fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 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 1 0 1 1 0 Rn Rd

FRSQRTE <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 1 sz 1 0 0 0 0 1 1 1 0 1 1 0 Rn Rd

FRSQRTE <V>d, <V>n

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 0 Rn Rd

FRSQRTE <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

FRSQRTE <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4S</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTypedefpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements - 1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, fpcr);
V[d] = 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 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 1 | 1 | Rn | Rd |

FRSQRTS <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

**Scalar single-precision and double-precision**

(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 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | Rsz | 1 | Rm | 1 | 1 | 1 | 1 | 1 | 1 | Rn | Rd |

FRSQRTS <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

**Vector half precision**

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 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 | Rm | 0 | 0 | 1 | 1 | 1 | 1 | Rn | Rd |

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

FQRSTS <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>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPRsqrtStepFused(element1, element2);
V[d] = result;
FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

( Armv8.2 )

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

FSQRT <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Single-precision and double-precision**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

FSQRT <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Assembler Symbols**

< Vd > Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

< T > For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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] = FPSqrt(element, FPCR[]);

V[d] = result;
FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 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>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftype</td>
<td>Rd</td>
</tr>
<tr>
<td>Rn</td>
<td></td>
</tr>
</tbody>
</table>

Half-precision (ftype == 11)  
(Armv8.2)

FSQRT <Hd>, <Hn>

Single-precision (ftype == 00)

FSQRT <Sd>, <Sn>

Double-precision (ftype == 01)

FSQRT <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '11' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\textbf{CheckFPAdvSIMDEnabled64}();

FPCRTypex fpcr = FPCR[];
boolean merge = \textbf{IsMerging}(fpcr);
bits(128) result = if merge then V[d] else \textbf{Zeros}();

bits(esize) operand = V[n];

\textbf{Elem}[\textit{result, 0, esize}] = \textbf{FPSqrt}(\textit{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 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 1 | 0 | 1 | Rn | Rd |

U

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 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 1 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |

U

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```assembly
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRType fpcr = FPCR[];
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```

External version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 Xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 1 0 ftype 1  Rm  0 1 1 1 0  Rn  Rd

Half-precision (ftype == 11)
(Armv8.2)

FSUB <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FSUB <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FSUB <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem(result, 0, esize) = FPSub(operand1, operand2, fpcr);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias \texttt{MOV (element)}.

\begin{verbatim}
INS \langle Vd \rangle.<Ts>[\langle index1 \rangle], \langle Vn \rangle.<Ts>[\langle index2 \rangle]

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;
\end{verbatim}

Assembler Symbols

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<Ts>} Is an element size specifier, encoded in "imm5":

\begin{verbatim}
\begin{array}{|c|}
\hline
imm5 & <Ts> \\
\hline
0000 & RESERVED \\
xxxx1 & B \\
xxxx10 & H \\
xx100 & S \\
x1000 & D \\
\hline
\end{array}
\end{verbatim}

\texttt{<index1>} Is the destination element index encoded in "imm5":

\begin{verbatim}
\begin{array}{|c|}
\hline
imm5 & <index1> \\
\hline
0000 & RESERVED \\
xxxx1 & imm5<4:1> \\
xxxx10 & imm5<4:2> \\
x1000 & imm5<4:3> \\
x1000 & imm5<4> \\
\hline
\end{array}
\end{verbatim}

\texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\texttt{<index2>} Is the source element index encoded in "imm5:imm4":

\begin{verbatim}
\begin{array}{|c|}
\hline
imm5 & <index2> \\
\hline
0000 & RESERVED \\
xxxx1 & imm4<3:0> \\
xxxx10 & imm4<3:1> \\
x1000 & imm4<3:2> \\
x1000 & imm4<3> \\
\hline
\end{array}
\end{verbatim}

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.
Operation

CheckFPAdvSIMDEnabled64();
bits(idxdsize) 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).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Ts>** Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<index>** Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

- **<R>** Is the width specifier for the general-purpose source register, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>

- **<n>** Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(128) result;

result = V[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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 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>]

```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  | 0  | 0  | 1  | 0  | 0  | 0  | 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>

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**

```haskell
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 = 3; selem = 1; // LD/ST1 (3 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

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |
| L | R | opcode |

8-bit (opcode == 000)

LD1 { <Vt>.B }[<index>], [<Xn|SP>]

16-bit (opcode == 010 && size == x0)

LD1 { <Vt>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 100 && size == 00)

LD1 { <Vt>.S }[<index>], [<Xn|SP>]

64-bit (opcode == 100 && S == 0 && size == 01)

LD1 { <Vt>.D }[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |
| L | R | opcode |
8-bit, immediate offset (Rm == 11111 && opcode == 000)
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1

8-bit, register offset (Rm != 11111 && opcode == 000)
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
         For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
         For the 32-bit variant: is the element index, encoded in "Q:S".
         For the 64-bit variant: is the element index, encoded in "Q".

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // 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 |

LD1R { <Vt>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

**Post-index**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | size | Rm | 1  | 1  | 0  | 0  | size | Rn | Rt |
| L  | R  | opcode | S |

Immediate offset (Rm == 11111)

LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

**Assembler Symbols**

- **<Vt>** Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<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.
- **<imm>** Is the post-index immediate offset, encoded in “size”: 
Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
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;
```

<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>
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selen-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 selen-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>0</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q 0 0 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1</td>
</tr>
</tbody>
</table>

LD2 { <Vt>, <T>, <Vt2>, <T> }, [<Xn|SP>]

t = Uint(Rt);
n = Uint(Rn);
m = integer UNKNOWN;
wback = FALSE;
tag_checked = wback || n != 31;

Post-index

<table>
<thead>
<tr>
<th>0</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q 0 0 1 1 0 0 0 1 1 0</td>
</tr>
</tbody>
</table>

Immediate offset (Rm == 1111)

LD2 { <Vt>, <T>, <Vt2>, <T> }, [<Xn|SP>], <imm>

Register offset (Rm != 1111)

LD2 { <Vt>, <T>, <Vt2>, <T> }, [<Xn|SP>], <Xm>

t = Uint(Rt);
n = Uint(Rn);
m = Uint(Rm);
wback = TRUE;
tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;  // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;  // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;  // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;  // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;  // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;  // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;  // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;  // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

```c
CheckFPAdvSIMEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    else
        SP[] = address + offs;
else
    X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LD2 (single structure)**

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |
| L | R | opcode |

**8-bit (opcode == 000)**

LD2 `{<Vt>.B, <Vt2>.B}[<index]), [<Xn|SP>]`

**16-bit (opcode == 010 && size == x0)**

LD2 `{<Vt>.H, <Vt2>.H}[<index]), [<Xn|SP>]`

**32-bit (opcode == 100 && size == 00)**

LD2 `{<Vt>.S, <Vt2>.S}[<index]), [<Xn|SP>]`

**64-bit (opcode == 100 && S == 0 && size == 01)**

LD2 `{<Vt>.D, <Vt2>.D}[<index]), [<Xn|SP>]`

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

### Post-index

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | | Rm | x | x | 0 | S | size | Rn | Rt |
| L | R | opcode |

LD2 (single structure)
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.


```plaintext
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q);    // S[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;
```

Shared Decode

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
        scale = 3;
    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;

integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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>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>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>size</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>
</tr>
<tr>
<td></td>
<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>
</tr>
</tbody>
</table>

LD2R { <Vt>,<T>, <Vt2>,<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

<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>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>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<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>
</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>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in “size”: 

LD2R
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
text
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
  integer datasize = if Q == '1' then 128 else 64;
  integer esize = 8 << scale;
text
```
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bvits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
            if memop == MemOp_LOAD then
                // insert into one lane of 128-bit register
                Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[t] = rval;
            else // memop == MemOp STORE
                // extract from one lane of 128-bit register
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
                offs = offs + ebytes;
                t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LD3 (multiple structures)**

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:

A is a packed array of 3-element structures. Each element is a 16-bit halfword.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

**No offset**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| L   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| L   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

**Immediate offset (Rm == 11111)**

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

**Register offset (Rm != 11111)**

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```
## Assembler Symbols

\(<Vt>\) Is the name of the first or only SIMD\&FP register to be transferred, encoded in the "Rt" field.

\(<T>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vt2>\) Is the name of the second SIMD\&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

\(<Vt3>\) 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

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3 (single structure)

Load single 3-element structure to one lane of three registers. This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rd</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8-bit (opcode == 001)


16-bit (opcode == 011 && size == x0)

LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 && S == 0 && size == 01)

LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n =UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rd</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)
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 == 0)
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset (Rm != 11111 && opcode == 011 && size == 0)
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 == 0)
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 0)
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
    scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOpSTORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
offs = Zeros();
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register
      Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
      V[t] = rval;
    else // memop == MemOp_STORE
      // extract from one lane of 128-bit register
      Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
      offs = offs + ebytes;
      t = (t + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opcode</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

LD3R { <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

<table>
<thead>
<tr>
<th></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>size</th>
<th>Rm</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opcode</td>
<td>S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Immediate offset (Rm == 11111)

LD3R { <Vt>,<T>, <Vt2>,<T>, <Vt3>,<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD3R { <Vt>,<T>, <Vt2>,<T>, <Vt3>,<T> }, [<Xn|SP>], <Xm>

integer t = 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.
<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>#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>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```plaintext
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
        scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

LD3R
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | size | Rn | Rt |
|    | L  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |


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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | size | Rm | 0  | 0  | 0  | 0  | size | Rn | Rt |
|    | L  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Immediate offset (Rm == 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 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;
```

LD4 (multiple structures)
Operation

```c
Operation?

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
<td>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>x</td>
<td>x</td>
<td>1</td>
<td>S</td>
<td>size</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>
</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 == 001)


16-bit (opcode == 011 && size == x0)


32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 && S == 0 && size == 01)


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>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rm</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>S</td>
<td>size</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>
</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, immediate offset (Rm == 11111 && opcode == 001)

8-bit, register offset (Rm != 11111 && opcode == 001)

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols
<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
    scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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**

```
0 | Q | 0 0 1 1 1 1 0 1 | 0 0 0 0 1 1 1 0 | size | Rn | Rt |
   | L  | R   | opcode | S  |
```

LD4R \{ \langle Vt \rangle, \langle T \rangle, \langle Vt2 \rangle, \langle T \rangle, \langle Vt3 \rangle, \langle T \rangle, \langle Vt4 \rangle, \langle T \rangle \}, [\langle Xn|SP \rangle]

```java
t = UInt(Rt);
n = UInt(Rn);
m = integer UNKNOWN;
bool wback = FALSE;
tag_checked = wback || n != 31;
```

**Post-index**

```
0 | Q | 0 0 1 1 0 1 1 1 | 1 1 0 8 6 5 4 3 2 1 0 | Rm | size | Rn | Rt |
   | L  | R   | opcode | S  |
```

Immediate offset (Rm == 11111)

LD4R \{ \langle Vt \rangle, \langle T \rangle, \langle Vt2 \rangle, \langle T \rangle, \langle Vt3 \rangle, \langle T \rangle, \langle Vt4 \rangle, \langle T \rangle \}, [\langle Xn|SP \rangle], <imm>

Register offset (Rm != 11111)

LD4R \{ \langle Vt \rangle, \langle T \rangle, \langle Vt2 \rangle, \langle T \rangle, \langle Vt3 \rangle, \langle T \rangle, \langle Vt4 \rangle, \langle T \rangle \}, [\langle Xn|SP \rangle], <Xm>

```java
t = UInt(Rt);
n = UInt(Rn);
m = UInt(Rm);
wback = TRUE;
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> Is the post-index immediate offset, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#4</td>
</tr>
<tr>
<td>01</td>
<td>#8</td>
</tr>
<tr>
<td>10</td>
<td>#16</td>
</tr>
<tr>
<td>11</td>
<td>#32</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
ingter Integer init_scale = UInt(opcode<2:1>);
ingter Integer scale = init_scale;
ingter Integer selem = UInt(opcode<0>:R) + 1;
ingter Boolean replicate = FALSE;
ingter 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;
ingter Integer datasize = if Q == '1' then 128 else 64;
ingter Integer esize = 8 << scale;
```

Page 994
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(128) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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.

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.

As 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.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
  Constraint c = 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();
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[ ];
else
  address = X[n];
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>0 0</td>
<td>1 1 0 0 1 1</td>
<td>#imm</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>0 1</td>
<td>1 1 0 1 1</td>
<td>#imm</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>0 1</td>
<td>1 0 1 1</td>
<td>#imm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDP (SIMD&FP)
32-bit (opc == 00)

LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]

64-bit (opc == 01)

LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm}>]

128-bit (opc == 10)

LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm}>]

boolean wback = FALSE;
boolean postindex = FALSE;

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP (SIMD&FP).

Assembler Symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.
For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.
For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.
For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

Operation

CheckFPAdvSIMEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  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**

| 31 30 29 28 27 26 25 24 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 | 0 | imm9 | 0 | 1 | Rn | Rt |

opc

8-bit (size == 00 && opc == 01)

LDR <Bt>, [<Xn|SP>], #<simm>

16-bit (size == 01 && opc == 01)

LDR <Ht>, [<Xn|SP>], #<simm>

32-bit (size == 10 && opc == 01)

LDR <St>, [<Xn|SP>], #<simm>

64-bit (size == 11 && opc == 01)

LDR <Dt>, [<Xn|SP>], #<simm>

128-bit (size == 00 && opc == 11)

LDR <Qt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
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 |
| size | 1 | 1 | 1 | 1 | 0 | 0 | x | 1 | 0 | imm9 | 1 | 1 | Rn | Rt |

opc
8-bit (size == 00 && opc == 01)
LDR <Bt>, [<Xn|SP>, #<simm>]

16-bit (size == 01 && opc == 01)
LDR <Ht>, [<Xn|SP>, #<simm>]

32-bit (size == 10 && opc == 01)
LDR <St>, [<Xn|SP>, #<simm>]

64-bit (size == 11 && opc == 01)
LDR <Dt>, [<Xn|SP>, #<simm>]

128-bit (size == 00 && opc == 11)
LDR <Qt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

<table>
<thead>
<tr>
<th>size</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm12</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td>Rt</td>
</tr>
</tbody>
</table>

LDR (immediate, SIMD&FP)
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)
Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
if !postindex then
  address = address + offset;
case memop of
  when MemOp_STORE
    data = V[t];
    Mem[address, datasize DIV 8, AccType_VEC] = data;
  when MemOp_LOAD
    data = Mem[address, datasize DIV 8, AccType_VEC];
    V[t] = data;
if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

32-bit (opc == 00)

LDR <St>, <label>

64-bit (opc == 01)

LDR <Dt>, <label>

128-bit (opc == 10)

LDR <Qt>, <label>

integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
  when '00' size = 4;
  when '01' size = 8;
  when '10' size = 16;
  when '11' UNDEFINED;

offset = SignExtend(imm19: '00', 64);

Assembler Symbols

<Dt> Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
.Qt Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<St> Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(64) address = PC[] + offset;
bits(size*8) data;
if HaveMTEExt() then
  SetTagCheckedInstruction(FALSE);
CheckFPAdvSIMDEnabled64();
data = Mem[address, size, AccType_VEC];
V[t] = data;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<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>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
</tr>
</tbody>
</table>

**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>

<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
inger = UInt(Rn);
t = UInt(Rt);
m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
datasize = 8 << scale;
tag_checked = memop != MemOp_PREFETCH;
```
Operation

bits(64) offset = \texttt{ExtendReg}(m, \texttt{extend\_type}, \texttt{shift});
if \texttt{HaveMTEExt}() then
  \texttt{SetTagCheckedInstruction}(\texttt{tag\_checked});
\texttt{CheckFPAdvSIMDEnabled64}();
bits(64) address;
bits(datasize) data;
if n == 31 then
  \texttt{CheckSPAlignment}();
  address = \texttt{SP}[];
else
  address = \texttt{X}[n];
address = address + offset;
case \texttt{memop} of
  when \texttt{MemOp\_STORE}
    data = \texttt{V}[t];
    \texttt{Mem}[address, datasize DIV 8, \texttt{AccType\_VEC}] = data;
  when \texttt{MemOp\_LOAD}
    data = \texttt{Mem}[address, datasize DIV 8, \texttt{AccType\_VEC}];
    \texttt{V}[t] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDUR (SIMD&FP)**

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1 1 1 1 0 0 x 1 0</th>
<th>imm9</th>
<th>0 0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 01)

LDUR <Bt>, [<Xn|SP>{} , #<simm>]

16-bit (size == 01 && opc == 01)

LDUR <Ht>, [<Xn|SP>{} , #<simm>]

32-bit (size == 10 && opc == 01)

LDUR <St>, [<Xn|SP>{} , #<simm>]

64-bit (size == 11 && opc == 01)

LDUR <Dt>, [<Xn|SP>{} , #<simm>]

128-bit (size == 00 && opc == 11)

LDUR <Qt>, [<Xn|SP>{} , #<simm>]

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<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
    SetTagCheckedInstruction(tag_checked);

CheckFPAAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAAlignment();
    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 \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | size | L | M | Rm | 0 | 0 | 0 | 0 | H | 0 | Rn | Rd |
    |     |
  o2

MLA <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');
\end{verbatim}

Assembler Symbols

\textbf{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\textbf{<T>} Is an arrangement specifier, encoded in "size:Q":
\begin{verbatim}
<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>
\end{verbatim}

\textbf{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\textbf{<Vm>} Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":
\begin{verbatim}
<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>
\end{verbatim}

Restricted to V0-V15 when element size <Ts> is \textit{H}.

\textbf{<Ts>} Is an element size specifier, encoded in "size":
\begin{verbatim}
<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>\textit{H}</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{<index>} Is the element index, encoded in "size:L:H:M":
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

MLS \(<Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]\)

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean 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

`CheckFPAdvSIMDEnabled64();`
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
ingternal element1;
ingternal 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.
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 \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.

\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');

\texttt{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 = \texttt{Elem}[operand1, e, esize];
  element2 = \texttt{Elem}[operand2, e, esize];
  product = (UInt(element1)*UInt(element2))<esize-1:0>;
  if sub_op then
    \texttt{Elem}[result, e, esize] = \texttt{Elem}[operand3, e, esize] - product;
  else
    \texttt{Elem}[result, e, esize] = \texttt{Elem}[operand3, e, esize] + product;
\end{verbatim}

\texttt{V}[d] = result;

---

Assembler Symbols

\begin{itemize}
  \item \texttt{<Vd>}, \texttt{<T>}, \texttt{<Vn>}, \texttt{<T>}, \texttt{<Vm>}, \texttt{<T>}
  \item \texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the \texttt{"Rd"} field.
  \item \texttt{<T>} Is an arrangement specifier, encoded in \texttt{"size:Q"}:
  \begin{tabular}{c|c|c}
    \texttt{size} & \texttt{Q} & \texttt{<T>} \\
    \hline
    00 & 0 & 8B \\
    00 & 1 & 16B \\
    01 & 0 & 4H \\
    01 & 1 & 8H \\
    10 & 0 & 2S \\
    10 & 1 & 4S \\
    11 & x & RESERVED
  \end{tabular}
  \item \texttt{<Vn>} Is the name of the first SIMD&FP source register, encoded in the \texttt{"Rn"} field.
  \item \texttt{<Vm>} Is the name of the second SIMD&FP source register, encoded in the \texttt{"Rm"} field.
\end{itemize}

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 = \texttt{Elem}[operand1, e, esize];
  element2 = \texttt{Elem}[operand2, e, esize];
  product = (UInt(element1)*UInt(element2))<esize-1:0>;
  if sub_op then
    \texttt{Elem}[result, e, esize] = \texttt{Elem}[operand3, e, esize] - product;
  else
    \texttt{Elem}[result, e, esize] = \texttt{Elem}[operand3, e, esize] + product;
\end{verbatim}

\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.
**MOV (scalar)**

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of **DUP (element)**. This means:

- The encodings in this description are named to match the encodings of **DUP (element)**.
- The description of **DUP (element)** gives the operational pseudocode for this instruction.

![Encoding Table]

### Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;V&gt;</td>
<td>Is the destination width specifier, encoded in &quot;imm5&quot;:</td>
</tr>
<tr>
<td>imm5</td>
<td>x0000: RESERVED</td>
</tr>
<tr>
<td></td>
<td>xxxx1: B</td>
</tr>
<tr>
<td></td>
<td>xxx10: H</td>
</tr>
<tr>
<td></td>
<td>xx100: S</td>
</tr>
<tr>
<td></td>
<td>x1000: D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;d&gt;</td>
<td>Is the number of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Vn&gt;</td>
<td>Is the name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;T&gt;</td>
<td>Is the element width specifier, encoded in &quot;imm5&quot;:</td>
</tr>
<tr>
<td>imm5</td>
<td>x0000: RESERVED</td>
</tr>
<tr>
<td></td>
<td>xxxx1: B</td>
</tr>
<tr>
<td></td>
<td>xxx10: H</td>
</tr>
<tr>
<td></td>
<td>xx100: S</td>
</tr>
<tr>
<td></td>
<td>x1000: D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;index&gt;</td>
<td>Is the element index encoded in &quot;imm5&quot;:</td>
</tr>
<tr>
<td>imm5</td>
<td>x0000: RESERVED</td>
</tr>
<tr>
<td></td>
<td>xxxx1: imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td></td>
<td>xxx10: imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td></td>
<td>xx100: imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td></td>
<td>x1000: imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

### Operation

The description of **DUP (element)** gives the operational pseudocode for this instruction.

### Operational information

If `PSTATE.DIT` is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register. This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (element). This means:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | imm5 | 0 | imm4 | 1 | Rn | Rd |

MOV <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]

is equivalent to

INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>x1000</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<index1> Is the destination element index encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in "imm5:imm4":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm4&lt;3:0&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm4&lt;3:1&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm4&lt;3:2&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm4&lt;3&gt;</td>
</tr>
</tbody>
</table>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

The description of INS (element) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the \textit{CPACR EL1}, \textit{CPTR EL2}, and \textit{CPTR EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of \textit{INS (general)}. This means:

- The encodings in this description are named to match the encodings of \textit{INS (general)}.
- The description of \textit{INS (general)} gives the operational pseudocode for this instruction.

\begin{verbatim}
MOV <Vd>.<Ts>[<index>], <R><n>

is equivalent to

INS <Vd>.<Ts>[<index>], <R><n>

and is always the preferred disassembly.
\end{verbatim}

\textbf{Assembler Symbols}

\begin{description}
\item [<Vd>] Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\item [<Ts>] Is an element size specifier, encoded in “imm5”:
\begin{verbatim}
imm5    <Ts>
\end{verbatim}
\begin{verbatim}
0000    RESERVED
xxx1    B
xxx10   H
xx100   S
x1000   D
\end{verbatim}
\item [<index>] Is the element index encoded in “imm5”:
\begin{verbatim}
imm5    <index>
\end{verbatim}
\begin{verbatim}
0000    RESERVED
xxx1    imm5<4:1>
xxx10   imm5<4:2>
xx100   imm5<4:3>
x1000   imm5<4>
\end{verbatim}
\item [<R>] Is the width specifier for the general-purpose source register, encoded in “imm5”:
\begin{verbatim}
imm5    <R>
\end{verbatim}
\begin{verbatim}
0000    RESERVED
xxx1    W
xxx10   W
xx100   W
x1000   X
\end{verbatim}
\item [<n>] Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.
\end{description}

\textbf{Operation}

The description of \textit{INS (general)} gives the operational pseudocode for this instruction.

\textbf{Operational information}

If \text{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.
MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of ORR (vector, register). This means:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
0 & Q & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & Rm & 0 & 0 & 0 & 1 & 1 & 1 & Rn & Rd \\
\hline
\end{array}
\]

\begin{tabular}{|c|}
\hline
size \\
\hline
\end{tabular}

\[
\text{MOV} \ <Vd>.<T>, \ <Vn>.<T>
\]

is equivalent to

\[
\text{ORR} \ <Vd>.<T>, \ <Vn>.<T>, \ <Vn>.<T>
\]

and is the preferred disassembly when \( Rm = Rn \).

Assembler Symbols

- \(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) Is an arrangement specifier, encoded in “Q”:

\[
\begin{array}{|c|}
\hline
0 & 8B \\
1 & 16B \\
\hline
\end{array}
\]

- \(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of \texttt{UMOV}. This means:

- The encodings in this description are named to match the encodings of \texttt{UMOV}.
- The description of \texttt{UMOV} gives the operational pseudocode for this instruction.

32-bit (Q == 0 && imm5 == xx100)

\texttt{MOV <Wd>, \langle Vn\rangle.S[\langle index\rangle]}

is equivalent to

\texttt{UMOV <Wd>, \langle Vn\rangle.S[\langle index\rangle]}

and is always the preferred disassembly.

64-reg,UMOV-64-reg (Q == 1 && imm5 == x1000)

\texttt{MOV <Xd>, \langle Vn\rangle.D[\langle index\rangle]}

is equivalent to

\texttt{UMOV <Xd>, \langle Vn\rangle.D[\langle index\rangle]}

and is always the preferred disassembly.

\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{<index>} For the 32-bit variant: is the element index encoded in "imm5<4:3>".
For the 64-reg,UMOV-64-reg variant: is the element index encoded in "imm5<4>".

\textbf{Operation}

The description of \texttt{UMOV} gives the operational pseudocode for this instruction.

\textbf{Operational information}

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

8-bit (op == 0 & cmode == 1110)

MOVI $<Vd>,<T>$, $#$<imm8>, {, LSL #0}

16-bit shifted immediate (op == 0 & cmode == 10x0)

MOVI $<Vd>,<T>$, $#$<imm8>{, LSL #<amount>}

32-bit shifted immediate (op == 0 & cmode == 0xx0)

MOVI $<Vd>,<T>$, $#$<imm8>{, LSL #<amount>}

32-bit shifting ones (op == 0 & cmode == 110x)

MOVI $<Vd>,<T>$, $#$<imm8>, MSL #<amount>

64-bit scalar (Q == 0 & op == 1 & cmode == 1110)

MOVI $<Dd>$, $#$<imm>

64-bit vector (Q == 1 & op == 1 & cmode == 1110)

MOVI $<Vd>,.2D$, $#$<imm>

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 '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;

    // FMODV Dn,#imm is in main FP instruction set
    if op == '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 ‘aaaaaaaaabbcccccdddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 \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}
MUL <Vd>..<T>, <Vn>..<T>, <Vm>..<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
\textbf{case size of}
\begin{itemize}
\item when '01' index = UInt(H:L:M); Rmhi = '0';
\item when '10' index = UInt(H:L); Rmhi = M;
\item otherwise UNDEFINED;
\end{itemize}

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;
\end{verbatim}

Assembler Symbols

\begin{itemize}
\item \textbf{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\item \textbf{<T>} Is an arrangement specifier, encoded in "size:Q":
\item \textbf{<Vm>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\item \textbf{<Vm>} Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":
\item \textbf{<Ts>} Is an element size specifier, encoded in "size":
\item \textbf{<index>} Is the element index, encoded in "size:L:H:M":
\end{itemize}
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    Elem[result, e, esize] = product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 | 1 | Rm | 1 | 0 | 0 | 1 | 1 | 1 | Rn | Rd |

MUL <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>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(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
  element1 = Elem(operand1, e, esize);
  element2 = Elem(operand2, e, esize);
  if poly then
    product = PolynomialMult(element1, element2)<esize-1:0>;
  else
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
  Elem[result, e, esize] = product;
V[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MVN

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of NOT. This means:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | Rn | Rd

MVN <Vd>.<T>, <Vn>.<T>
is equivalent to
NOT <Vd>.<T>, <Vn>.<T>

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&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

The description of NOT gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MVNI**

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | d  | e  | f  | g  | h  | Rd |

**16-bit shifted immediate (cmode == 10x0)**

```ml
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

**32-bit shifted immediate (cmode == 0xx0)**

```ml
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

**32-bit shifting ones (cmode == 110x)**

```ml
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;

```ml
ImmediateOp operation;
case cmode:op of
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '1101' operation = ImmediateOp_MVNI;
  when '1011' operation = ImmediateOp_MVNI;
  when '1101' operation = ImmediateOp_MVNI;
  when '1110' operation = ImmediateOp_MVNI;
  when '11111' // FMOV Dn,#imm is in main FP instruction set
  if Q == '0' then UNDEFINED;
  operation = ImmediateOp_MOVI;

imm64 = AdvSIMDEmm(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":
  ```ml
  Q  <T>
  0  4H
  1  8H
  ```
  For the 32-bit variant: is an arrangement specifier, encoded in "Q":
  ```ml
  Q  <T>
  0  2S
  1  4S
  ```
- `<imm8>` Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".
- `<amount>` For the 16-bit shifted immediate variant: is the shift amount encoded in "cmode<1>".
defaulting to 0 if LSL is omitted.
For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.
For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

<table>
<thead>
<tr>
<th>cmode&lt;0&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
b bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 0</td>
<td>1 0 0 0 0 0 1 1 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NEG <V><d>, <V><n>

```java
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 1 0 1 1 1 0</td>
<td>1 0 0 0 0 0 1 1 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NEG <Vd>.<T>, <Vn>.<T>

```java
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{Vn}> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(\text{datasize}) \text{operand} = \text{V}[n];
bits(\text{datasize}) \text{result};
\text{integer element};

\text{for } e = 0 \text{ to elements-1 }
    \text{element} = \text{SInt(\text{Elem}[\text{operand}, e, esize]);}
    \text{if neg then}
        \text{element} = -\text{element};
    \text{else}
        \text{element} = \text{Abs(\text{element});}
    \text{Elem}[\text{result, e, esize}] = \text{element}<\text{esize}-1:0>;
\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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
NOT

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MVN.

```
NOT <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

**Assembler Symbols**

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>
```

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = NOT(element);

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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>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>
</table>
|   | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | Rm | 0  | 0  | 1  | 1  | Rn | 0  | 0  | 1  | 1  | Rd |    |    |    |    |    |    |    |    |    | size

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

```c
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 \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.

32-bit (\texttt{cmode} == 0xx1)

ORR \texttt{<Vd>.<T>}, \#\texttt{<imm8>\{, LSL \texttt{<amount>}\}}

Assembler Symbols

\texttt{<Vd>} Is the name of the SIMD&FP register, encoded in the "Rd" field.

\texttt{<T>} For the 16-bit variant: is an arrangement specifier, encoded in "Q":

\begin{table}[h]
\centering
\begin{tabular}{ll}
\texttt{Q} & \texttt{<T>} \\
\hline
0 & 4H \\
1 & 8H \\
\end{tabular}
\end{table}  

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

\begin{table}[h]
\centering
\begin{tabular}{ll}
\texttt{Q} & \texttt{<T>} \\
\hline
0 & 2S \\
1 & 4S \\
\end{tabular}
\end{table}  

\texttt{<imm8>} Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

\texttt{<amount>} For the 16-bit variant: is the shift amount encoded in "\texttt{cmode}\langle1\rangle":

\begin{table}[h]
\centering
\begin{tabular}{ll}
\texttt{cmode}\langle1\rangle & \texttt{<amount>} \\
\hline
0 & 0 \\
1 & 8 \\
\end{tabular}
\end{table}  

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in "\texttt{cmode}\langle2:1\rangle":
<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>size</td>
</tr>
</tbody>
</table>

ORR <Vd>..<T>, <Vn>..<T>, <Vm>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector)</td>
<td>Rm == Rn</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
result = operand1 OR operand2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 1 | 1 | 1 | Rn | Rd |

PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```plaintext
type integer d = UInt(Rd);
type integer n = UInt(Rn);
type integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
type integer esize = 8 << UInt(size);
type integer datasize = if Q == '1' then 128 else 64;
type 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**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over \{0, 1\}.*

The PMULL instruction extracts each source vector from the lower half of each source register, while the PMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 1 | 1 | 0 | 0 | 0 | Rn | Rd |

PMULL\{2\} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !HaveBit128PMULLExt() then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>2</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>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
**Operation**

```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.
RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The results are rounded. For truncated results, see ADDHN.

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![hexadecimal diagram]

RADDHN(2) $<Vd>,<Tb>,<Vn>,<Ta>,<Vm>$

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>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[absent]</td>
<td>[present]</td>
</tr>
</tbody>
</table>

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Tb>$ Is an arrangement specifier, encoded in "size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8B</td>
<td>16B</td>
<td>4H</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.
**RAX1**

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when **FEAT_SHA3** is implemented.

**Advanced SIMD**  
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | Rm |
|    |    |    |    |    |    |    |    |    |    |
| 1  | 0  | 0  | 0  | 1  | 1  | Rn |
|    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    | Rd |

`RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D`

if !`HaveSHA3Ext()` then UNDEFINED;
integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;
integer m = `UInt(Rm)`;

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext```
`AArch64.CheckFPAdvSIMDEnabled()`;

`bits(128) Vm = V[m];`
`bits(128) Vn = V[n];`
`V[d] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));`
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RBIT (vector)

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |     |
|     Rn |     Rd |

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.
REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 Q 0 0 1 1 1 0 size 1 0 0 0 0 0 0 0 1 1 0 Rn Rd
```

REV16 `<Vd>..<T>`, `<Vn>..<T>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=size:   B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q".

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>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

```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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   1|   0|

Rn | Rd |
---|----|
```

REV32 <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;
end case;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Assembler Symbols

<table>
<thead>
<tr>
<th>Motor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;Vd&gt;</strong></td>
<td>The name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td><strong>&lt;T&gt;</strong></td>
<td>An arrangement specifier, encoded in &quot;size:Q&quot;.</td>
</tr>
<tr>
<td><strong>&lt;Vn&gt;</strong></td>
<td>The name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
</tbody>
</table>
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.
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.

### Assembly Symbols

- **Q**
  - 0 [absent]
  - 1 [present]

- **<Vd>**
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Tb>**
  - Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>**
  - Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- **<Ta>**
  - Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<shift>**
  - Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

```assembly
RSHRN(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');
```
### immh | <shift>
--- | ---
0000 | **SEE Advanced SIMD modified immediate**
0001 | (16-UInt(immh:immb))
001x | (32-UInt(immh:immb))
01xx | (64-UInt(immh:immb))
1xxx | RESERVED

## Operation

```c
CheckFPAdvSIMDEnabled64();
bv(bits(datasize*2) operand = V[n];
bv(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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | size|    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 0   |    |    |    |    | U  | Rd |

RSUBHN(2) <Vd>, <Tb>, <Vn>, <Ta>, <Vm>.

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();}

\begin{verbatim}
bits(2*datasize) operand1 = \texttt{V}[n];
b...
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.

| 31 30 29 28 27 26 25 24 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 execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to 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 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 destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register, while the SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
U 0 Q 0 0 1 1 1 0 | size | 1 | Rm | 0 1 0 1 0 0 | Rn | Rd

SABAL{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 accumulate = (op == '0');
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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.

![Binary Representation]

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];
bite(datasize) operand2 = V[m];
bite(datasize) result;
integer element1;
integer element2;
bite(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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction writes the vector to the lower half of the destination register and clears the upper half, while the SABDL2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</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>Q</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

SABDL[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 accumulate = (op == '0');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
b 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.
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.

```
31  30  29  28  27  26  25  24  23  22  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 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Rn | Rd |
  U | op |

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

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | Q  | 0  | 1  | 1  | 0  | size | 1  | Rm | 0  | 0  | 0  | 0  | 0  | Rn | 0  | Rd |    |
|    |    |    |    |    |    |      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SADDL(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

\[
\text{CheckFPAdvSIMDEnabled64}();
\]
\[
\text{bits(}\text{datasize}\text{)} \text{ operand1} = \text{Vpart}[n, \text{part}];
\]
\[
\text{bits(}\text{datasize}\text{)} \text{ operand2} = \text{Vpart}[m, \text{part}];
\]
\[
\text{bits(}\text{2\*datasize}\text{)} \text{ result};
\]
\[
\text{integer element1};
\]
\[
\text{integer element2};
\]
\[
\text{integer sum};
\]
\[
\text{for e = 0 to elements-1}
\]
\[
\text{element1} = \text{Int(}\text{Elem}([\text{operand1, e, esize}], \text{unsigned});}
\]
\[
\text{element2} = \text{Int(}\text{Elem}([\text{operand2, e, esize}], \text{unsigned});}
\]
\[
\text{if sub_op then}
\]
\[
\text{sum} = \text{element1 - element2};
\]
\[
\text{else}
\]
\[
\text{sum} = \text{element1 + element2};
\]
\[
\text{Elem}\{\text{result, e, 2\*esize}\} = \text{sum}<2\text{\{esize-1:0\}>};
\]
\[
\text{V[d]} = \text{result};
\]

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.
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 \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 0 | 0 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Rn | Rd | op
\end{verbatim}

SADDLP \texttt{<Vd>.<Ta>}, \texttt{<Vn>.<Tb>}

\begin{verbatim}
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 \texttt{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');
\end{verbatim}

**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>\texttt{&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>\texttt{&lt;Tb&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0  Q  0  0  1  1  1  0  size  1  1  0  0  0  0  0  1  1  1  0  Rd  Rn
U
```

SADDLV <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');
```

Assembler Symbols

- `<V>` Is the destination width specifier, encoded in “size”:
  ```plaintext
  | size | <V> |
  |------------------|
  | 00 | H |
  | 01 | S |
  | 10 | D |
  | 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”:
  ```plaintext
  | 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

```plaintext
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 | 0 | size | 1 | Rm | 0 | 0 | 0 | 1 | 0 | 0 | Rn | Rd |
| U | o1 |

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

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</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.

<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

```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.
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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

SCVT $<V><d>$, $<V><cn>$, #$<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 datsize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

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 datsize = if Q == '1' then 128 else 64;
integer elements = datsize 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”:

SCVT (vector, fixed-point)
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_AAdvanced SIMD modified immediate</td>
</tr>
<tr>
<td>0010</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0011</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>0100</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>0000</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>0010</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>0010</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>0100</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding);
V[d] = result;
```
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

U

0 1 | 0 1 1 1 1 0 | 0 1 1 1 1 0 0 1 1 1 0 1 1 0 | Rn | Rd

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

U

0 1 | 0 1 1 1 1 0 | 0 sz 1 0 0 0 0 1 1 1 0 1 1 0 | Rn | Rd

SCVTF <V>d>, <V>n

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

U

0 0 | 0 1 1 1 1 0 | 0 1 1 1 1 0 0 1 1 1 0 1 1 0 | Rn | Rd

SCVTF (vector, integer)
SCVTFT \(<V_d>.<T>, <V_n>.<T>\)

if \(!\text{HaveFP16Ext}()\) then UNDEFINED;

\[
\begin{align*}
\text{integer } d & = \text{UInt}(\text{Rd}); \\
\text{integer } n & = \text{UInt}(\text{Rn}); \\
\text{integer } e\text{size} & = 16; \\
\text{integer } \text{datasize} & = \text{if } Q == '1' \text{ then 128 else 64}; \\
\text{integer } \text{elements} & = \text{datasize} \div \text{esize}; \\
\text{boolean } \text{unsigned} & = (U == '1');
\end{align*}
\]

**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  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rn |  |  |  |  |  |  |  | Rd |  |
| U  |

SCVTFT \(<V_d>.<T>, <V_n>.<T>\)

\[
\begin{align*}
\text{integer } d & = \text{UInt}(\text{Rd}); \\
\text{integer } n & = \text{UInt}(\text{Rn}); \\
\text{if } sz:Q == '1' \text{ then UNDEFINED}; \\
\text{integer } e\text{size} & = 32 \text{ << UInt}(sz); \\
\text{integer } \text{datasize} & = \text{if } Q == '1' \text{ then 128 else 64}; \\
\text{integer } \text{elements} & = \text{datasize} \div \text{esize}; \\
\text{boolean } \text{unsigned} & = (U == '1');
\end{align*}
\]

**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>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<\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 half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;\text{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;\text{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

\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(\text{datasize}) \text{operand} = V[n];}

\texttt{FPCRType \ fpcr = FPCR[];}
\texttt{boolean \ merge = \text{elements} == 1 && \text{IsMerging(\text{fpcr});}}
\texttt{bits(128) \text{result} = \text{if} \ merge \text{then} \ V[d] \text{else} \text{Zeros();}}

\texttt{FPRounding \ rounding = FPRoundingMode(\text{fpcr});}
\texttt{bits(\text{esize}) \text{element;}}
\texttt{for \ e = 0 \text{to} \text{elements-1}}
\hspace{1em} \texttt{\text{element} = \text{Elem}[\text{operand, e, esize}];}
\hspace{1em} \texttt{\text{Elem}[\text{result, e, esize}] = \text{FixedToFP}(\text{element, 0, unsigned, \text{fpcr, rounding});}}

\texttt{V[d] = \text{result;}}
SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| sf  | 0   | 0   | 1   | 1   | 1   | 0   | ftype| 0   | 0   | 0   | 0   | 1   | 0   | scale|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| rmode| opcode |
```
32-bit to half-precision (sf == 0 & ftype == 11)  
(Armv8.2)

SCVTF <Hd>, <Wn>, #<fbits>

32-bit to single-precision (sf == 0 & ftype == 00)
SCVTF <Sd>, <Wn>, #<fbits>

32-bit to double-precision (sf == 0 & ftype == 01)
SCVTF <Dd>, <Wn>, #<fbits>

64-bit to half-precision (sf == 1 & ftype == 11)  
(Armv8.2)

SCVTF <Hd>, <Xn>, #<fbits>

64-bit to single-precision (sf == 1 & ftype == 00)
SCVTF <Sd>, <Xn>, #<fbits>

64-bit to double-precision (sf == 1 & ftype == 01)
SCVTF <Dd>, <Xn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  end;

if sf == '0' & scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR[0]);

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**

```plaintext
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, FALSE, fpcr, rounding);
V[d] = fltval;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
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
CheckFPAdvSIMDEnabled64();

FPCRType fpkr = FPCR[];
boolean merge = IsMerging(fpkr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, FALSE, fpkr, rounding);
V[d] = fltval;
```
**SDOT (by element)**

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an **optional** instruction. From Armv8.4 it is mandatory for all implementations to support it.

`ID_AAA64ISAAR0_EL1`.DP indicates whether this instruction is supported.

### Vector (Armv8.2)

|       | Q | 0 | 1 | 1 | 1 | L | M | Rd | Rn | H | 0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|---|---|---|---|---|---|---|----|----|---|---|---|---|---|---|---|---|---|---|---|---|---|
| U     |   |   |   |   |   |   |   |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**SDOT** `<Vd>,<Ta>, <Vn>,<Tb>, <Vm>.4B[<index>]`

```plaintext
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);
```

```plaintext
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”:
  ```plaintext
<table>
<thead>
<tr>
<th>Q</th>
<th>Ta</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”:
  ```plaintext
<table>
<thead>
<tr>
<th>Q</th>
<th>Tb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>
  ```
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- `<index>` Is the element index, encoded in the "H:L" fields.
Operation

\textbf{CheckFPAdvSIMDEnabled64()},

\textbf{bits}(datasize) operand1 = \textbf{V}[n];
\textbf{bits}(128) operand2 = \textbf{V}[m];
\textbf{bits}(datasize) result = \textbf{V}[d];
for e = 0 to elements - 1
  integer res = 0;
  integer element1, element2;
  for i = 0 to 3
    if signed then
      element1 = \textbf{SInt}(\textbf{Elem}[operand1, 4*e+i, esize DIV 4]);
      element2 = \textbf{SInt}(\textbf{Elem}[operand2, 4*index+i, esize DIV 4]);
    else
      element1 = \textbf{UInt}(\textbf{Elem}[operand1, 4*e+i, esize DIV 4]);
      element2 = \textbf{UInt}(\textbf{Elem}[operand2, 4*index+i, esize DIV 4]);
    end if
    res = res + element1 * element2;
    \textbf{Elem}[result, e, esize] = \textbf{Elem}[result, e, esize] + res;
\textbf{V}[d] = result;
**SDOT (vector)**

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an **OPTIONAL** instruction. From Armv8.4 it is mandatory for all implementations to support it.

**ID AA64ISARO_EL1**. DP indicates whether this instruction is supported.

### Vector (Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0  | 0  | 1 | 1 | 1 | 0 | size | 0 | Rm | 1 | 0 | 0 | 1 | 0 | 1 | Rn | Rd |

**SDOT** `<Vd>,<Ta>, <Vn>,<Tb>, <Vm>,<Tb>`

```plaintext
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

- **<Vd>** is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- **<Ta>** is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

- **<Vn>** is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Tb>** is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- **<Vm>** is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;

V[d] = result;
SHA1C

SHA1 hash update (choose).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SHA1C \(<Q_d>, \langle S_n>, <V_m>\).4S

\[
\text{integer } d = \text{UInt}(Rd); \\
\text{integer } n = \text{UInt}(Rn); \\
\text{integer } m = \text{UInt}(Rm); \\
\text{if !HaveSHA1Ext() then UNDEFINED;}
\]

Assembler Symbols

\(<Q_d>\) Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
\(<S_n>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
\(<V_m>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

\[
\text{AARCH64.CheckFPAdvSIMDEnabled()};
\]

\[
\begin{align*}
\text{bits(128)} & \quad X = V[d]; \\
\text{bits(32)} & \quad Y = V[n]; \quad \text{// Note: 32 not 128 bits wide} \\
\text{bits(128)} & \quad W = V[m]; \\
\text{bits(32)} & \quad t;
\end{align*}
\]

for e = 0 to 3
\[
\begin{align*}
\quad & \quad t = \text{SHAChoose}(X<63:32>, X<95:64>, X<127:96>); \\
\quad & \quad Y = Y + \text{ROL}(X<31:0>, 5) + t + \text{Elem}[W, e, 32]; \\
\quad & \quad X<63:32> = \text{ROL}(X<63:32>, 30); \\
\quad & \quad <Y, X> = \text{ROL}(Y:X, 32); \\
\quad & \quad V[d] = X;
\end{align*}
\]

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.
SHA1H

SHA1 fixed rotate.

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & Rn & Rd
\end{array}
\]

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).

<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>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SHA1M <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 = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
  Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
  X<63:32> = ROL(X<63:32>, 30);
  <Y, X> = ROL(Y:X, 32);
V[d] = X;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1P

SHA1 hash update (parity).

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SHA1P <Qd>, <Sn>, <Vm>.45

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n];    // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;
for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1SU0

SHA1 schedule update 0.

```
0 1 0 1 1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0
```

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.

integer \( d = \text{UInt}(\text{Rd}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
if \(!\text{HaveSHA1Ext}()\) then UNDEFINED;

Assembler Symbols

\(<\text{Vd}>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
\(<\text{Vn}>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

\[ \text{AArch64.CheckFPAdvSIMDEnabled}() ; \]
\[ \text{bits}(128) \text{ operand1} = \text{V}[d] ; \]
\[ \text{bits}(128) \text{ operand2} = \text{V}[n] ; \]
\[ \text{bits}(128) \text{ result} ; \]
\[ \text{bits}(128) \text{ T} = \text{operand1 \ EOR \ LSR(operand2, 32)} ; \]
\[ \text{result}<31:0> = \text{ROL(T<31:0>, 1)} ; \]
\[ \text{result}<63:32> = \text{ROL(T<63:32>, 1)} ; \]
\[ \text{result}<95:64> = \text{ROL(T<95:64>, 1)} ; \]
\[ \text{result}<127:96> = \text{ROL(T<127:96>, 1) \ EOR \ ROL(T<31:0>, 2)} ; \]
\[ \text{V}[d] = \text{result} ; \]

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.
SHA256H2

SHA256 hash update (part 2).

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SHA256H

SHA256 hash update (part 1).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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</td>
</tr>
</tbody>
</table>

SHA256H \(<Q_d>, <Q_n>, <V_m>\).4S

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
if !\text{HaveSHA256Ext}() then UNDEFINED;

Assembler Symbols

\(<Q_d>\) Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
\(<Q_n>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
\(<V_m>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

\text{AArch64.CheckFPAdvSIMDEnabled}();

\text{bits}(128) \text{ result};
result = \text{SHA256hash}(V[d], V[n], V[m], \text{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.
SHA256SU0

SHA256 schedule update 0.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  |

SHA256SU0 <Vd>.4S, <Vn>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();
bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0>:operand1<127:32>;
bits(32) elt;
for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA256SU1

SHA256 schedule update 1.

<table>
<thead>
<tr>
<th>SHA256SU1</th>
<th>.4S, .4S, .4S</th>
</tr>
</thead>
</table>

integer \(d\) = UInt(Rd);
integer \(n\) = UInt(Rn);
integer \(m\) = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

**Assembler Symbols**

\(<Vd>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

**Operation**

\(\text{AArch64.CheckFPAdvSIMDEnabled}()\);

\(\text{bits(128)} \operatorname{operand1} = V[d];\)
\(\text{bits(128)} \operatorname{operand2} = V[n];\)
\(\text{bits(128)} \operatorname{operand3} = V[m];\)
\(\text{bits(128)} \operatorname{result};\)
\(\text{bits(128)} T0 = \operatorname{operand3}[31:0]:\operatorname{operand2}[127:32];\)
\(\text{bits(64)} T1;\)
\(\text{bits(32)} \operatorname{elt};\)

\(T1 = \operatorname{operand3}[127:64];\)
for \(e = 0\) to \(1\)
\(\quad \operatorname{elt} = \text{Elem}[T1, e, 32];\)
\(\quad \operatorname{elt} = \text{ROR}(\operatorname{elt}, 17) \text{EOR} \text{ROR}(\operatorname{elt}, 19) \text{EOR} \text{LSR}(\operatorname{elt}, 10);\)
\(\quad \operatorname{elt} = \operatorname{elt} + \text{Elem}[\operatorname{operand1}, e, 32] + \text{Elem}[T0, e, 32];\)
\(\quad \text{Elem}[\operatorname{result}, e, 32] = \operatorname{elt};\)

\(T1 = \operatorname{result}[63:0];\)
for \(e = 2\) to \(3\)
\(\quad \operatorname{elt} = \text{Elem}[T1, e-2, 32];\)
\(\quad \operatorname{elt} = \text{ROR}(\operatorname{elt}, 17) \text{EOR} \text{ROR}(\operatorname{elt}, 19) \text{EOR} \text{LSR}(\operatorname{elt}, 10);\)
\(\quad \operatorname{elt} = \operatorname{elt} + \text{Elem}[\operatorname{operand1}, e, 32] + \text{Elem}[T0, e, 32];\)
\(\quad \text{Elem}[\operatorname{result}, e, 32] = \operatorname{elt};\)

\(V[d] = \operatorname{result};\)

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 `FEAT_SHA512` is implemented.

Advanced SIMD
(Armv8.2)

<table>
<thead>
<tr>
<th>SHA512H2</th>
<th>&lt;Qd&gt;, &lt;Qn&gt;, &lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0</td>
<td>Rm 1 0 0 0 0 1</td>
</tr>
</tbody>
</table>

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();
b(128) Vtmp;
b(64) NSigma0;
b(128) X = V[n];
b(128) Y = V[m];
b(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.
SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when **FEAT_SHA512** is implemented.

### Advanced SIMD (Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**SHA512H**

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

```asm
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>, 18) EOR ROR(Y<127:64>, 41);
Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);
tmp = Vtmp<127:64> + Y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND Y<127:64>) EOR (NOT(tmp) AND X<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + W<63:0>);
V[d] = Vtmp;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register. This instruction is implemented only when FEAT_SHA512 is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  |

SHA512SU0 <Vd>.2D, <Vn>.2D

if !HaveSHA512Ext () then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

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 FEAT_SHA512 is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

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;
b bits(128) Vtmp;
b bits(128) X = V[n];
b bits(128) Y = V[m];
b 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<63:8> = W<63:8> + sig1 + Y<63:8>;
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 SHRADD.

Depending on the settings in the CPACR_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 size 1 Rm 0 0 0 0 1 Rn Rd
```

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 | immb | 0  | 1  | 0  | 1  | 0 | 1 | Rn | Rd |

SHL <V><d>, <V><cn>, #<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  | Q | 0  | 0  | 1  | 1  | 1  | 1  | 0  | != 0000 | immb | 0  | 1  | 0  | 1  | 0 | 1 | Rn | Rd |

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

<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>
</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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

\[\text{SHLL}(2) \ <Vd>.<Ta>, \ <Vn>.<Tb>, \ #:<shift>\]

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd) ; \\
\text{integer } n & = \text{UInt}(Rn) ; \\
\text{if size } &= \ '11' \ \text{then UNDEFINED}; \\
\text{integer esize} & = 8 \ll \text{UInt}(\text{size}); \\
\text{integer datasize} & = 64 ; \\
\text{integer part} & = \text{UInt}(Q) ; \\
\text{integer elements} & = \text{datasize DIV esize}; \\
\text{integer shift} & = \text{esize}; \\
\text{boolean unsigned} & = \text{FALSE} ; \quad // \text{Or TRUE without change of functionality}
\end{align*}
\]

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”:

\[
\begin{array}{c|c}
Q & 2 \\
- & - \\
0 & \text{[absent]} \\
1 & \text{[present]} \\
\end{array}
\]

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

\(<Ta>\) Is an arrangement specifier, encoded in “size”:

\[
\begin{array}{c|c}
\text{size} & <Ta> \\
00 & 8H \\
01 & 4S \\
10 & 2D \\
11 & \text{RESERVED} \\
\end{array}
\]

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the “Rn” field.

\(<Tb>\) Is an arrangement specifier, encoded in “size:Q”:

\[
\begin{array}{c|c|c}
\text{size} & Q & <Tb> \\
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & \text{RESERVED} \\
\end{array}
\]

\(<\text{shift}>\) Is the left shift amount, which must be equal to the source element 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

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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 \textit{RSHRN}.

The \textit{RSHRN} instruction writes the vector to the lower half of the destination register and clears the upper half, while the \textit{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 \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.

```
| 0 | Q | 0 | 1 | 1 | 1 | 0 | != 0000 | immh | 1 | 0 | 0 | 0 | 1 | Rd |
```

\textbf{SHRN(2)} \langle Vd \rangle, \langle Tb \rangle, \langle Vn \rangle, \langle Ta \rangle, \#<shift>

\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 part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
\end{verbatim}

\textbf{Assembler Symbols}

\begin{itemize}
    \item \textbf{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}
    | 0 | 1 |
    | [absent] | [present] |
    \end{verbatim}

    \item \textbf{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
    \item \textbf{<Tb>} Is an arrangement specifier, encoded in "immh:Q":
    \begin{verbatim}
    \begin{array}{c|c}
    \hline
    \text{immh} & \text{Q} \\
    \hline
    0000 & x \\
    0001 & 0 \\
    0001 & 1 \\
    001x & 0 \\
    001x & 1 \\
    01xx & 0 \\
    01xx & 1 \\
    1xxx & x \\
    \hline
    \end{array}
    \end{verbatim}

    \item \textbf{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.
    \item \textbf{<Ta>} Is an arrangement specifier, encoded in "immh":
    \begin{verbatim}
    \begin{array}{c|c}
    \hline
    \text{immh} & \text{<Ta>} \\
    \hline
    0000 & \text{SEE Advanced SIMD modified immediate} \\
    0001 & 8H \\
    001x & 4S \\
    01xx & 2D \\
    1xxx & \text{RESERVED} \\
    \hline
    \end{array}
    \end{verbatim}

    \item \textbf{<shift>} Is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":
\end{itemize}
### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
for e = 0 to elements-1
  element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
  Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
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.

![Shift Left and Insert Example](image)

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```plaintext
SLI <V><d>, <V><cn>, #<shift>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

**Vector**

```plaintext
SLI <Vd>.<T>, <Vn>.<T>, #<shift>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;`
Assembler Symbols

Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xx</td>
<td>D</td>
</tr>
</tbody>
</table>

Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the 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(Oesizesizesize, shift);
bits(esize) shifted;
for e = 0 to elements-1
  shifted = LSL(Elem[operand, e, esize], shift);
  Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD (Armv8.2)

```

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):ROL(Vm<95:64>, 15):ROL(Vm<63:32>, 15));

for i = 0 to 3
  if i == 3 then
    result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
    result<(32*i)+31:(32*i)> = result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>, 15) EOR ROL(result<(32*i)+31:(32*i)>, 23);
  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.
SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  |

SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):ROL(Vm<95:64>, 7):ROL(Vm<63:32>, 7):ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**SM3SS1**

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when **FEAT_SM3** is implemented.

**Advanced SIMD**
(Armv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | Ra | Rn | Rd |

SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S
```

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Va>` Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

**Operation**

AArch64.CheckFPAdvSIMDEnabled();

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) Va = V[a];
Vd<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
Vd<95:0> = Zeros();
V[d] = Vd;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, \( V_d \), 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, \( V_n \), 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, \( V_m \).

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when \texttt{FEAT_SM3} is implemented.

Advanced SIMD
(Armv8.2)

<table>
<thead>
<tr>
<th>Rm</th>
<th>imm2</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0 0 1 0</td>
<td>[imm2]</td>
<td>1 0</td>
<td>Rd</td>
</tr>
</tbody>
</table>

\[
\text{SM3TT1A} \ <V_d>.45, \ <V_n>.45, \ <V_m>.S[<imm2>]
\]

if \(!\text{HaveSM3Ext}()\) then UNDEFINED;
integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Rm) \);
integer \( i = \text{UInt}(\text{imm2}) \);

Assembler Symbols

- \(<V_d>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- \(<V_n>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- \(<V_m>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- \(<\text{imm2}>\) Is a 32-bit element indexed out of \(<V_m>\), encoded in "imm2".

Operation

\[
\text{AArch64.CheckFPAdvSIMDEnabled}();
\]

\[
\begin{align*}
\text{bits}(128) \ V_m &= V[m]; \\
\text{bits}(128) \ V_n &= V[n]; \\
\text{bits}(128) \ V_d &= V[d]; \\
\text{bits}(32) \ W_{j\text{Prime}}; \\
\text{bits}(128) \ \text{result}; \\
\text{bits}(32) \ TT1; \\
\text{bits}(32) \ SS2;
\end{align*}
\]

\[
W_{j\text{Prime}} = \text{Elem}[V_m, i, 32];
\]

\[
\begin{align*}
SS2 &= V_n<127:96> \ \text{EOR} \ \text{ROL}(V_d<127:96>, 12); \\
TT1 &= V_d<63:32> \ \text{EOR} \ (V_d<127:96> \ \text{EOR} \ V_d<95:64>); \\
TT1 &= (TT1+V_d<31:0>+SS2+W_{j\text{Prime}})<31:0>; \\
\text{result}<31:0> &= \text{Vd}<63:32>; \\
\text{result}<63:32> &= \text{ROL}(V_d<95:64>, 9); \\
\text{result}<95:64> &= \text{Vd}<127:96>; \\
\text{result}<127:96> &= TT1; \\
V[d] &= \text{result};
\end{align*}
\]

Operational information

If \text{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM3TT1B

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 FEAT_SM3 is implemented.

**Advanced SIMD**
(Armv8.2)

|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 31    | 30    | 29    | 28    | 27    | 26    | 25    | 24    | 23    | 22    | 21    | 20    | 19    | 18    | 17    | 16    | 15    | 14    | 13    | 12    | 11    | 10    | 9     | 8     | 7     | 6     | 5     | 4     | 3     | 2     | 1     | 0     |
| 1     | 1     | 0     | 1     | 1     | 0     | 0     | 0     | 0     | 1     | 0     | 1     | 0     | imm2  | 1     | Rm    |

SM3TT1B \(<Vd>.45, <Vn>.45, <Vm>.S[<imm2>]\)

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

**Assembler Symbols**

- \(<Vd>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- \(<Vn>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- \(<Vm>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- \(<imm2>\) Is a 32-bit element indexed out of \(<Vm>\), encoded in "imm2".

**Operation**

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = (Vd<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 response of this instruction to 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 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 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector,Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when FEAT_SM3 is implemented.

### Advanced SIMD (Armv8.2)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rm</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

SM3TT2A `<Vd>.4S, <Vn>.4S, <Vm>.S<imm2>`

```java
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

```java
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Rm | 1  | 0  | imm2 | 1  | 1  | Rn | Rd |

SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S<imm2>

```plaintext
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

```plaintext
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

SM4E Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register. This instruction is implemented only when FEAT_SM4 is implemented.

Advanced SIMD
(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1</td>
</tr>
</tbody>
</table>

SM4E <Vd>.4S, <Vn>.4S

if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vn = V[n];
bits(32) intval;
bits(8) sboxout;
bits(128) roundresult;
bits(32) roundkey;

roundkey = V[d];
for index = 0 to 3
    intval = Elem[Vn, index, 32];
    roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    intval = intval EOR roundresult<31:0>;

roundresult<31:0> = roundresult<63:32>;
roundresult<63:32> = roundresult<95:64>;
roundresult<95:64> = roundresult<127:96>;
roundresult<127:96> = intval;
V[d] = roundresult;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SM4 is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | Rm | 1 | 1 | 0 | 0 | 1 | 0 | Rn |  | Rd |

SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(32) intval;
bits(8) sboxout;
bits(128) result;
bits(32) const;
bits(128) roundresult;

roundresult = V[n];
for index = 0 to 3
  const = Elem[Vm, index, 32];
  intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
  for i = 0 to 3
    Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
  intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
  intval = intval EOR roundresult<31:0>;
roundresult<31:0> = roundresult<63:32>;
roundresult<63:32> = roundresult<95:64>;
roundresult<95:64> = roundresult<127:96>;
roundresult<127:96> = intval;
V[d] = roundresult;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
  element1 = Int(Elem(operand1, e, esize), unsigned);
  element2 = Int(Elem(operand2, e, esize), unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
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.

$$\text{integer } d = \text{UInt}(Rd);$$
$$\text{integer } n = \text{UInt}(Rn);$$
$$\text{integer } m = \text{UInt}(Rm);$$
$$\text{if size }= '11' \text{ then UNDEFINED;}$$
$$\text{integer esize }= 8 \times \text{UInt}(size);$$
$$\text{integer datasize }= \text{if } Q = '1' \text{ then 128 else 64;}$$
$$\text{integer elements }= \text{datasize} \div \text{esize};$$

$$\text{boolean unsigned }= (U = '1');$$
$$\text{boolean minimum }= (o1 = '1');$$

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMAXV

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 1 1 1 0 1 1 1 0 0 0 0 1 0 1 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 1 0
size  0 1 1 0 0 1 1 0 0 0 0 1 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0
Rn   Rd
U     op

SMAXV <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

- `<V>` Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<T>` Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
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.
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 \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.

```
0 1 1 0 1 1 1 0 | size | Rd
0 1 1 0 1 1 0 1 | Rm
```

\textbf{Assembler Symbols}

- \textbf{<Vd>}: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \textbf{<T>}: Is an arrangement specifier, encoded in "size:Q":
  
  \begin{tabular}{c|c|c}
    size & Q & <T> \\
    \hline
    00 & 0 & 8B \\
    00 & 1 & 16B \\
    01 & 0 & 4H \\
    01 & 1 & 8H \\
    10 & 0 & 2S \\
    10 & 1 & 4S \\
    11 & x & RESERVED \\
  \end{tabular}

- \textbf{<Vn>}: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \textbf{<Vm>}: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

\textbf{Operation}

\begin{verbatim}
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;
\end{verbatim}

\textbf{Operational information}

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
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 \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}
integer d = UInt(Rd);
ingter n = UInt(Rn);
ingter m = UInt(Rm);
if size == '11' then UNDEFINED;
ingter esize = 8 \times UInt(size);
ingter datasize = if Q == '1' then 128 else 64;
ingter elements = datasize \div esize;
boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
\end{verbatim}

### Assembler Symbols

\begin{itemize}
  \item <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
  \item <T> Is an arrangement specifier, encoded in “size:Q”:
  \begin{table}[h]
  \begin{tabular}{c|c|c}
  \hline
  size & Q & <T> \\
  \hline
  00 & 0 & 8B \\
  00 & 1 & 16B \\
  01 & 0 & 4H \\
  01 & 1 & 8H \\
  10 & 0 & 2S \\
  10 & 1 & 4S \\
  11 & x & RESERVED \\
  \hline
  \end{tabular}
  \end{table}
  \item <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
  \item <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\end{itemize}

### Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
ingter element1;
ingter element2;
ingter 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;
\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.
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.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 Q 0 0 1 1 1 0 | size | 1 1 0 0 0 1 1 0 1 0 1 0 | Rd | Rd |

SMINV <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

<V> Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
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.

![Hexadecimal code snippet](image)

**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>Ta</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>Tb</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.

<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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   0 | Q  | 0  | 1  | 1  | 0  | size | 1  | Rm | 1  | 0  | 0  | 0  | 0  | Rn | Rd | o1 |
```

**SMLAL2** `<Vd>`, `<Vn>`, `<Vm>`, `<Ta>`, `<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”:

```
Q
0 [absent]
1 [present]
```

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in “size”:

```
size  <Ta>
00   8H
01   4S
10   2D
11   RESERVED
```

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in “size:Q”:

```
size  Q  <Tb>
00   0   8B
00   1   16B
01   0   4H
01   1   8H
10   0   2S
10   1   4S
11   x   RESERVED
```

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```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.

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
<th>H</th>
<th>L:1:0</th>
<th>M:1:0</th>
<th>Rm:1:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28:27</td>
<td>26:25</td>
<td>24:23</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19:18</td>
<td>17:16</td>
<td>15:14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10:9</td>
<td>8:7</td>
<td>6:5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

SMLSL\{2\} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
</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>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>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESERVED</td>
<td>0:Rm</td>
<td>M:Rm</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>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESERVED</td>
<td>H</td>
<td>S</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>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESERVED</td>
<td>H:L:M</td>
<td>H:L</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register, while the SMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
   U   |   Rd   |   Rn   |   size   |   Rm   |
0 1 1 0 1 1 0 | 0 0 1 0 1 0 | 0 0 |

SMLSL2 {2} <Vd>, <Ta>, <Vn>, <Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td></td>
<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) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
```

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SMMLA (vector)**

Signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an **OPTIONAL** instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_EL1*.I8MM indicates whether this instruction is supported.

**Vector**
(Armv8.6)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |
| B  | Rm| 1 | 0 | 1 | 0 | 0 | 1 | Rn | Rd |

**SMMLA** $<\text{Vd}>.4S,<\text{Vn}>.16B,<\text{Vm}>.16B$

if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);

**Assembler Symbols**

$<\text{Vd}>$ Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

$<\text{Vn}>$ 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**

CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];

$V[d]$ = MatMulAdd(addend, operand1, operand2, FALSE, FALSE);
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 \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{tabular}{|l|l|l|}
\hline
0 & Q & 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ imm5 & 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ Rn \ Rd \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ imm5 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ imm5 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ \hline
\end{tabular}

### 32-bit (Q == 0)

\texttt{SMOV <Wd>, <Vn>.<Ts>[<index>]}

### 64-reg, SMOV-64-reg (Q == 1)

\texttt{SMOV <Xd>, <Vn>.<Ts>[<index>]}

\begin{itemize}
  \item integer \( d = \texttt{UInt}(Rd) \);
  \item integer \( n = \texttt{UInt}(Rn) \);
  \item integer \( size \);
  \item case \( Q:iimm5 \) of
    \begin{itemize}
      \item when \'xxxxx1' size = 0; // SMOV [WX]d, Vn.B
      \item when \'xxxx10' size = 1; // SMOV [WX]d, Vn.H
      \item when \'1xx100' size = 2; // SMOV Xd, Vn.S
      \item otherwise UNDEFINED;
    \end{itemize}
  \item integer \( idxdsize = \text{if} \ imm5<4> == '1' \ \text{then} \ 128 \ \text{else} \ 64; \)
  \item integer \( \text{index} = \texttt{UInt}(imm5<4:size+1>); \)
  \item integer \( \text{esize} = 8 \ll size; \)
  \item integer \( \text{datasize} = \text{if} \ Q == '1' \ \text{then} \ 64 \ \text{else} \ 32; \)
\end{itemize}

#### Assembler Symbols

\begin{itemize}
  \item \( <Wd> \) \hspace{1cm} Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
  \item \( <Xd> \) \hspace{1cm} Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
  \item \( <Vn> \) \hspace{1cm} Is the name of the SIMD&FP source register, encoded in the "Rn" field.
  \item \( <Ts> \) \hspace{1cm} For the 32-bit variant: is an element size specifier, encoded in "imm5":
    \begin{itemize}
      \item \( \text{imm5} \)<\( <Ts> \)
      \begin{itemize}
        \item xxx00 \hspace{0.5cm} \text{RESERVED}
        \item xxx1 \hspace{0.5cm} B
        \item xxx10 \hspace{0.5cm} H
      \end{itemize}
    \end{itemize}
    For the 64-reg, SMOV-64-reg variant: is an element size specifier, encoded in "imm5":
    \begin{itemize}
      \item \( \text{imm5} \)<\( <Ts> \)
      \begin{itemize}
        \item xx000 \hspace{0.5cm} \text{RESERVED}
        \item xxx1 \hspace{0.5cm} B
        \item xxx10 \hspace{0.5cm} H
        \item xx100 \hspace{0.5cm} S
      \end{itemize}
    \end{itemize}
  \item \( <\text{index}> \) \hspace{1cm} For the 32-bit variant: is the element index encoded in "imm5":
    \begin{itemize}
      \item \( \text{imm5} \)<\( <\text{index}> \)
      \begin{itemize}
        \item xxx00 \hspace{0.5cm} \text{RESERVED}
        \item xxx1 \hspace{0.5cm} \text{imm5<4:1>}
        \item xxx10 \hspace{0.5cm} \text{imm5<4:2>}
      \end{itemize}
    \end{itemize}
    For the 64-reg, SMOV-64-reg variant: is the element index encoded in "imm5":
\end{itemize}
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
</tbody>
</table>

**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.
SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register, while the SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

**SMULL2** <Vd>,<Ta>, <Vn>,<Tb>, <Vm>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</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 &lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</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";
Restricted to V0-V15 when element size $<Ts>$ is H.

$<Ts>$
Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<\text{index}>$
Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\text{index}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

\[
\text{element2} = \text{Int}(\text{Elem}[\text{operand2, index, esize}], \text{unsigned});
\]
for e = 0 to elements-1
  \[
  \text{element1} = \text{Int}(\text{Elem}[\text{operand1, e, esize}], \text{unsigned});
  \]
  \[
  \text{product} = (\text{element1} * \text{element2}) < 2*\text{esize}-1:0>;
  \]
  \[
  \text{Elem}[\text{result, e, 2*\text{esize}}] = \text{product};
  \]

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.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
</tbody>
</table>
```

SMULL{2} <Vd>, <Ta>, <Vn>, <Vm>, <Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

**Assembler Symbols**

| 2 | Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:
<table>
<thead>
<tr>
<th></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>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SQABS**

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \texttt{FPSR.QC} is set.

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}

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\texttt{SQABS \langle V \rangle \langle d \rangle, \langle V \rangle \langle n \rangle}

integer \( d = \texttt{UInt}(Rd) \);
integer \( n = \texttt{UInt}(Rn) \);

integer esize = 8 << \texttt{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>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\texttt{SQABS \langle Vd \rangle \langle T \rangle, \langle Vn \rangle \langle T \rangle}

integer \( d = \texttt{UInt}(Rd) \);
integer \( n = \texttt{UInt}(Rn) \);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
inherent elements = datasize DIV esize;
boolean neg = (U == '1');

**Assembler Symbols**

\texttt{<V>} Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>\texttt{&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>

\texttt{<d>} Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<n>} Is the number of the SIMD&FP source register, encoded in the "Rn" field.

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<T>} Is an arrangement specifier, encoded in “size:Q”: 
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>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**

`CheckFPAdvSIMDEnabled64();`

```c
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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

```
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 0 | 0 | 1 | 1 | Rn | Rd |
```

SQADD `<V><d>`, `<V><n>`, `<V><m>`

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `integer esize = 8 << UInt(size);`
- `integer datasize = esize;`
- `integer elements = 1;`
- `boolean unsigned = (U == '1');`

### Vector

```
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 0 | 0 | 1 | 1 | Rn | Rd |
```

SQADD `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<T>`

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `if size:Q == '110' then UNDEFINED;`
- `integer esize = 8 << UInt(size);`
- `integer datasize = if Q == '1' then 128 else 64;`
- `integer elements = datasize DIV esize;`
- `boolean unsigned = (U == '1');`

### Assembler Symbols

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the “Rd” field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the “Rn” field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the “Rm” field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the “Rd” field.
- `<T>` Is an arrangement specifier, encoded in “size:Q”: 
\[
\begin{array}{c|c|c}
\text{size} & \text{Q} & \langle T \rangle \\
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & 0 & \text{RESERVED} \\
11 & 1 & 2D \\
\end{array}
\]

\text{<Vn>} \quad \text{Is the name of the first SIMD\&FP source register, encoded in the "Rn" field.}

\text{<Vm>} \quad \text{Is the name of the second SIMD\&FP source register, encoded in the "Rm" field.}

**Operation**

\begin{verbatim}
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;
\end{verbatim}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMLAL, SQDMLAL2 (by element)

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLAL instruction extracts vector elements from the lower half of the first source register, while the SQDMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 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 |

SQDMLAL \(<Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]\)

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

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 | 0 | 1 | 1 | 1 | size | L | M | Rm | 0 | 0 | 1 | 1 | H | 0 | Rn | Rd | o2 |

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 |

SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

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 | 0 | 1 | 1 | 1 | size | L | M | Rm | 0 | 0 | 1 | 1 | H | 0 | Rn | Rd | o2 |

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 |

SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bite Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

**Assembler Symbols**

<table>
<thead>
<tr>
<th>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;<strong>Ta</strong>&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;<strong>Tb</strong>&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;<strong>Va</strong>&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;<strong>Vb</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>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.

Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
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;
```

Copyright © 2010-2020 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 hit FPSR.QC is set.

The SQDMLAL instruction extracts each source vector from the lower half of each source register, while the SQDMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

![Scalar encoding](image)

- **SQDMLAL** \(<Va><d>, <Vb><n>, <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

![Vector encoding](image)

- **SQDMLAL[2]** \(<Vd>, <Ta>, <Vn>, <Tb>, <Vm>, <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 elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

### Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
```

SQDMLAL, SQDMLAL2 (vector)
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 1 0 1 1 1 1 1 | size | L | M | Rm | 0 1 1 1 | H | 0 | Rn | Rd |
| o2 

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 0 0 0 1 1 1 1 | size | L | M | Rm | 0 1 1 1 | H | 0 | Rn | Rd |
| o2 

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

Assembler Symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper
   64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta>  Is an arrangement specifier, encoded in “size”:  

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb>  Is an arrangement specifier, encoded in “size:Q”:  

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va>  Is the destination width specifier, encoded in “size”:  

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<Vb>  Is the source width specifier, encoded in “size”:  

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>  Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:  

SQDMLSL, SQDMLSL2 (by element)
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;
bbits(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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMLSL, SQDMLSL2 (vector)**

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The **SQDMLSL** instruction extracts each source vector from the lower half of each source register, while the **SQDMLSL2** instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
 0  1  0  1  1  1  1  0  |  size  | 1  | Rm  | 1  0  1  1  0  0  | Rn  |  Rd  | 0 1
```

```
SQDMLSL <Va><d>, <Vb><n>, <Vb><m>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');

### Vector

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
 0  1  0  1  1  1  1  0  |  size  | 1  | Rm  | 1  0  1  1  0  0  | Rn  |  Rd  | 0 1
```

```
SQDMLSL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer 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>
</tbody>
</table>
```
<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va>  Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb>  Is the source width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m>  Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
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 SQRDMLUH.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and 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 | 0 | H | 0 | Rn | Rd |

op

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 1 | size | L | M | Rm | 1 | 1 | 0 | 0 | H | 0 | Rn | Rd |

op

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

`CheckFPAdvSIMDEnabled64();`

bits(datasize) operand1 = V[n];
bits(idxsiz) 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 *SQDMULH*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the *CPACR_EL1, CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
| U  |

SQDMULH <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');

**Vector**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
| U  |

SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts the first source vector from the lower half of the first source register, while the SQDMULL2 instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<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>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQDMULL <Va><d>, <Vb><n>, <Vm><Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

Vector

<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>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 &lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</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 Q &lt;Tb&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>

<Va>  

Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size &lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</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 &lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<n>  

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  

Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size &lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</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 &lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<index>  

Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size &lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>
Operation

`CheckFPAdvSIMDEnabled64();`

bits(datasize) operand1 = \textit{Vpart}[n, part];
bits(idxdsize) operand2 = \textit{V}[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

\begin{verbatim}
  element2 = \textit{SInt}(\textit{Elem}[operand2, index, esize]);
  for e = 0 to elements-1
    element1 = \textit{SInt}(\textit{Elem}[operand1, e, esize]);
    \textbf{(product, sat)} = \textit{SignedSatQ}(2 * element1 * element2, 2 * esize);
    \textit{Elem}[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';
\end{verbatim}

\textbf{V}[d] = result;
SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register, while the SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 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   1   1   0   1   0   0 | Rm  | 1   |
| 0   0   1   1   0   0   1   0 | Rd  |     |

SQDMULL <Va><d>, <Vb><n>, <Vb><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 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   1   1   0   1   0   0 | Q   | 1   |
| 0   0   1   1   0   0   1   0 | Rm  |     |
| 0   0   1   1   0   0   1   0 | Rd  |     |

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 elements = datasize DIV esize;

Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Q 2

<absent> | [present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":
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>Tb</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</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>x</td>
</tr>
</tbody>
</table>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

< Vm > Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<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>Vb</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

Is the number of the first SIMD&FP source register, encoded in the "Rn" 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.

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 \texttt{FPSR.QC} is set.

Depending on the settings in the \texttt{CPACR\_EL1, 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}

### Scalar

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>1</td>
</tr>
<tr>
<td>Rd</td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Symbols**

\(<V>\) \hspace{1cm} \text{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>

\(<d>\) \hspace{1cm} \text{Is the number of the SIMD&FP destination register, encoded in the "Rd" field.}

\(<n>\) \hspace{1cm} \text{Is the number of the SIMD&FP source register, encoded in the "Rn" field.}

\(<Vd>\) \hspace{1cm} \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}

\(<T>\) \hspace{1cm} \text{Is an arrangement specifier, encoded in “size:Q”:}

```python
def SQNEG_V<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');
```

```python
def SQNEG_V<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');
```
<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;
boolean sat;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMMLAH (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)*

```
0 1 1 1 1 1 1 1 size L M Rm 1 1 0 1 H 0 Rn Rd
```

```
SQRDMMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>] 
```

if !HaveQRQDMMLAHExe() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

**Vector**
*(Armv8.1)*

```
0 O|1 0 1 1 1 1 size L M Rm 1 1 0 1 H 0 Rn Rd
```

SQRDMMLAH (by element)
SQRDMLAH \(<Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]\)

if \(!\text{HaveQRDMLAHExt}()\) 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

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
ineger element2;
ineger element3;
ineger 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;
```
SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded. If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(Armv8.1)

31 30 29 28 27 26 25 24 23 22 21 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 | 0 | Rm | 1 | 0 | 0 | 0 | 1 | Rn | Rd |

SQRDMLAH <V><d>, <V><n>, <V><m>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector
(Armv8.1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 0 | size | 0 | Rm | 1 | 0 | 0 | 0 | 1 | Rn | Rd |

SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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  | 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  |

SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 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  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
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

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
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 1 0 size 0 | Rm | 1 0 0 0 1 1 | Rn | Rd | S
```

SQRDMLSH <V<d>, <V<n>, <V<m>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector
(Armv8.1)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 1 | 1 | 1 | 1 | 1 | size 0 | Rm | 1 0 0 0 1 1 | Rn | Rd | S
```

SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see 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  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   | size | L | M | Rm | 1 | 1 | 0 | 1 | H | 0 | Rn | Rd |
| op |
```

SQRDMULH <V>d>, <V>n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean round = (op == '1');
```

Vector

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | Q | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| op |
```

SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = datasize DIV esize;
boolean round = (op == '1');
```
### Assembler Symbols

- **<V>** is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<d>** is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- **<n>** is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Vd>** is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<T>** is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Vm>** is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

- **<Ts>** is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<index>** is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

\texttt{CheckFPAdvSIMDEnabled64()};

\texttt{bits(datasize) operand1 = V[n];}
\texttt{bits(idxdsize) operand2 = V[m];}
\texttt{bits(datasize) result;}
\texttt{integer round\_const = if round then 1 << (esize - 1) else 0;}
\texttt{integer element1;}
\texttt{integer element2;}
\texttt{integer product;}
\texttt{boolean sat;}

\texttt{element2 = \texttt{SInt(Element[operand2, index, esize]);}}
\texttt{for e = 0 to elements-1}
\hspace{1em} \texttt{element1 = \texttt{SInt(Element[operand1, e, esize]);}}
\hspace{1em} \texttt{product = (2 * element1 * element2) + round\_const;}
\hspace{1em} // The following only saturates if element1 and element2 equal \(-2^{(esize-1)}\)
\hspace{1em} \texttt{(Element[result, e, esize], sat) = \texttt{SignedSatQ(product >> esize, esize);}}
\hspace{1em} \texttt{if sat then FPSR.QC = '1';}

\texttt{V[d] = result;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rm</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 elements = 1;
boolean rounding = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rm</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 elements = datasize DIV esize;
boolean rounding = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see SQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 1 1 1 0 | size | 1 |
| 0 1 0 1 1 1 | Rn |
| Rd |
| U |
| R |
| S |

SQRSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 1 1 1 0 | size | 1 |
| 0 1 0 1 1 1 | Rn |
| Rd |
| U |
| R |
| S |

SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SQRSHRN.

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|
| 0 1 1 1 1 1 1 1 | != 0000 | immb | 1 0 0 1 | 1 | Rn | Rd | op |

SQRSHRN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immb == '0000' then UNDEFINED;
if immb<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immb);
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 1 1 1 1 1 1 1 | != 0000 | immb | 1 0 0 1 | 1 | Rn | Rd | op |

SQRSHRN2 <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>

<\textit{Vd}> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<\textit{Tb}> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>\text{immh}</th>
<th>\text{Q}</th>
<th>\text{&lt;Tb&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>x</td>
<td>\text{RESERVED}</td>
</tr>
</tbody>
</table>

<\textit{Vn}> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<\textit{Ta}> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>\text{immh}</th>
<th>\text{&lt;Ta&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\text{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>\text{RESERVED}</td>
</tr>
</tbody>
</table>

<\textit{Vb}> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>\text{immh}</th>
<th>\text{&lt;Vb&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\text{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>\text{RESERVED}</td>
</tr>
</tbody>
</table>

<\textit{d}> Is the number of the SIMD&FP destination register, in the “Rd” field.

<\textit{Va}> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>\text{immh}</th>
<th>\text{&lt;Va&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\text{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>\text{RESERVED}</td>
</tr>
</tbody>
</table>

<\textit{n}> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<\textit{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>\text{immh}</th>
<th>\text{&lt;shift&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\text{RESERVED}</td>
</tr>
<tr>
<td>0001</td>
<td>\text{16-UInt(immh:immb)}</td>
</tr>
<tr>
<td>001x</td>
<td>\text{32-UInt(immh:immb)}</td>
</tr>
<tr>
<td>01xx</td>
<td>\text{64-UInt(immh:immb)}</td>
</tr>
<tr>
<td>1xxx</td>
<td>\text{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>\text{immh}</th>
<th>\text{&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>\text{16-UInt(immh:immb)}</td>
</tr>
<tr>
<td>001x</td>
<td>\text{32-UInt(immh:immb)}</td>
</tr>
<tr>
<td>01xx</td>
<td>\text{64-UInt(immh:immb)}</td>
</tr>
<tr>
<td>1xxx</td>
<td>\text{RESERVED}</td>
</tr>
</tbody>
</table>
Operation

\textbf{CheckFPAdvSIMDEnabled64}();

\begin{verbatim}
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;
\end{verbatim}
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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0 != 0000</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>immh</td>
</tr>
</tbody>
</table>

SQRSHRUN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 1 0 != 0000</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>immh</td>
</tr>
</tbody>
</table>

SQRSHRUN2 <Vd>, <Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then \text{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 1214: Encoding of Specialized SIMD Instructions

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Vd&gt;</th>
<th>&lt;Tb&gt;</th>
<th>&lt;Vn&gt;</th>
<th>&lt;Ta&gt;</th>
<th>&lt;Vb&gt;</th>
<th>&lt;d&gt;</th>
<th>&lt;Va&gt;</th>
<th>&lt;n&gt;</th>
<th>&lt;shift&gt;</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>
</tr>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</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>
</tr>
<tr>
<td>0000</td>
<td>8H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>4S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>2D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>RESERVED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**<Vd>** is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Tb>** is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>** is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<Ta>** is an arrangement specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vb>** is the destination width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<d>** is the number of the SIMD&FP destination register, in the "Rd" field.

**<Va>** is the source width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<n>** is the number of the first SIMD&FP source register, encoded in the "Rn" field.

**<shift>** for the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16- UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32- UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64- UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

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;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSHL (immediate)**

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see **UQRSHL**.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | != 0000 | immh | 0 | 1 | 1 | 0 | 1 | Rn | Rd |

U

immh

op

SQSHL <V><d>, <V><cn>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 ** HighestSetBit**(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

**Vector**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | != 0000 | immh | 0 | 1 | 1 | 0 | 1 | Rn | Rd |

U

immh

op

SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 ** HighestSetBit**(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
Assembler Symbols

<v> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<t> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</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>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 the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
ingeger 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;
**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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 size</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 1 0 size</td>
</tr>
</tbody>
</table>

**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":

<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>;
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \textit{UQRSHL}.

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}.

**Scalar**

\begin{verbatim}
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;
end case;
\end{verbatim}

**Vector**

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then \textit{SFE(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;
end case;
\end{verbatim}
Assembler Symbols

<\text{V}>  
Is a width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>\text{&lt;V&gt;0}</th>
<th>\text{&lt;V&gt;0000}</th>
<th>\text{RESERVED}</th>
<th>\text{&lt;V&gt;0001}</th>
<th>B</th>
<th>\text{&lt;V&gt;001x}</th>
<th>H</th>
<th>\text{&lt;V&gt;01xx}</th>
<th>S</th>
<th>\text{&lt;V&gt;1xxx}</th>
<th>D</th>
</tr>
</thead>
</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{Vd}>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\text{T}>  
Is an arrangement specifier, encoded in "immh:Q":

| immh  | \text{<T>Q} | \text{<T>0000} x | \text{SEE Advanced SIMD modified immediate} | \text{<T>0001} 0 | B | \text{<T>001x} 1 | 16B | \text{<T>01xx} 0 | 4H | \text{<T>01xx} 1 | 8H | \text{<T>1xxx} 0 | 2S | \text{<T>1xxx} 1 | 4S | \text{<T>1xxx} 0 | RESERVED | \text{<T>1xxx} 1 | 2D |

<\text{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":

| immh  | \text{<shift>} | \text{<shift>0000} RESERVED | \text{<shift>0001} ( UInt(immh:immb)-8) | \text{<shift>001x} ( UInt(immh:immb)-16) | \text{<shift>01xx} ( UInt(immh:immb)-32) | \text{<shift>1xxx} ( UInt(immh:immb)-64) |

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":

| immh  | \text{<shift>} | \text{<shift>0000} RESERVED | \text{<shift>0001} ( UInt(immh:immb)-8) | \text{<shift>001x} ( UInt(immh:immb)-16) | \text{<shift>01xx} ( UInt(immh:immb)-32) | \text{<shift>1xxx} ( UInt(immh:immb)-64) |

Operation

\text{CheckFPAdvSIMDEnabled64}();

\text{bits(datasize) operand = V[n];}

\text{bits(datasize) result;}

\text{integer element; boolean sat;}

\text{for e = 0 to elements-1}

\quad \text{element = Int(Elem[operand, e, esize], src_unsigned) \ll shift;}

\quad \text{(Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);}

\quad \text{if sat then FPSR.QC = '1';}

\quad \text{V[d] = result;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 **SQRSHRN**.

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
0 1 1 1 1 1 1 0 | != 0000 | immh | 1 0 0 1 0 1 | Rn | Rd
U
```

```
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

```
0 | 0 | 1 1 1 1 1 0 | != 0000 | immh | 1 0 0 1 0 1 | Rn | Rd
U
```

```
SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #=><shift>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then **SFE(asimdimm)**;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(0);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '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>&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

\[ \text{CheckFPAdvSIMDEnabled64}(); \]

bits(datasize*2) operand = \( V[n] \);
bits(datasize) result;
integer round_const = if round then \((1 << (\text{shift} - 1))\) 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';
\]

\( V_{\text{part}}[d, \text{part}] = \text{result}; \)
SQSHRUN, SQSHRUN2

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see SQRSRUN.

The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
<table>
<thead>
<tr>
<th>31</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>1</td>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>immh</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

immh op
```

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>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>1</td>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>immh</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

immh op
```

SQSHRUN[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><code>&lt;Q&gt;</code></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 x</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0 8B</td>
</tr>
<tr>
<td>0001</td>
<td>1 16B</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x 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><code>&lt;Ta&gt;</code></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><code>&lt;Vb&gt;</code></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><code>&lt;Va&gt;</code></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><code>&lt;shift&gt;</code></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><code>&lt;shift&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

`CheckFPAdvSIMDEnabled64();`

bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
   element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
   (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
   if sat then FPSR.QC = '1';

Vpart[d, part] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQSUB

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 0 1 | 1 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 0 1 0 1 | 1 | Rn | Rd |

U
```

SQSUB <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector

```
| 31 30 29 28 27 26 25 24 23 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
```

SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

SQSUB
<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
void CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | 1 0 | Rn | Rd |
```

\( SQXTN <Vb><d>, <Va><n> \)

```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 | 1 0 | Rn | Rd |
```

\( SQXTN{2} <Vd>.<Tb>, <Vn>.<Ta> \)

```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":

```
Q 2
0 [absent]
1 [present]
```

\( <Vd> \) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
**<Tb>** Is an arrangement specifier, encoded in “size:Q”:

<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>x</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<Ta>** Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

**<Vb>** Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

**<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Va>** Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

**<n>** Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQXTUN, SQXTUN2

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQXTUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |

SQXTUN{2} <Vd>.<Td>, <Vn>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
in teger esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

Assembler Symbols

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 2 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 2 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

0 [absent]
1 [present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Tb> Is an arrangement specifier, encoded in "size: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":

SQXTUN, SQXTUN2
<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>Q</code></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>

**<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><code>size</code></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>

**<Vb>** Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>&lt;Vb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Va>** Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>&lt;Va&gt;</code></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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SRHADD**

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see **SHADD**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

---

### Assembler Symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
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 © 2010-2020 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 example of shift right operation]

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

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 1 0 != 0000 immh 0 1 0 0 0 1 Rn Rd

SRI \langle V \rangle\langle d \rangle, \langle V \rangle\langle n \rangle, \#\langle shift \rangle

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - \texttt{UInt}(immh:immb);
\end{verbatim}

\textbf{Vector}

\begin{verbatim}
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 1 1 1 0 != 0000 immh 0 1 0 0 0 1 Rn Rd

SRI \langle Vd \rangle\langle T \rangle, \langle Vn \rangle\langle T \rangle, \#\langle shift \rangle

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

if immh == '0000' then \texttt{SEE(asimdimm)};
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << \texttt{HighestSetBit}(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - \texttt{UInt}(immh:immb);
\end{verbatim}
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 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 = 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.
**SRSHL**

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see **SSHL**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 30 29 28 27 26 25 24 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 | 0 1 0 1 0 1 | 0 1 0 1 0 1 | Rn | Rd |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | | | size | 1 | Rm | 0 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |
| U | R | S |

SRSHL \(<V><d>, <V><n>, <V><m>\)

```
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 | size | Rm | 0 1 0 1 0 1 | 0 1 0 1 0 1 | Rn | Rd |
|---|---|---|---|---|---|---|---|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | | size | 1 | Rm | 0 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |
| U | R | S |

SRSHL \(<Vd>,<T>, <Vn>,<T>, <Vm>,<T>\)

```
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>0</td>
</tr>
</tbody>
</table>

- **<d>** is the number of the SIMD&FP destination register, in the "Rd" field.
- **<n>** is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- **<m>** is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
        else
            Elem[result, e, esize] = element<esize-1:0>;
    V[d] = result;
```
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 **SSHR**.

Depending on the settings in the *CPACR_EL1, CPTR_EL2, and 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
 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**

```plaintext
 integer d = UInt(Rd);
 integer n = UInt(Rn);
 if immh == '0000' then **SEE(asimdimm)**;
 if immh<3>:Q == '1' then UNDEFINED;
 integer esize = 8 << **HighestSetBit**(immh);
 integer datasize = if Q == '1' then 128 else 64;
 integer elements = datasize DIV esize;
 integer shift = (esize * 2) - UInt(immh:immb);
 boolean unsigned = (U == '1');
 boolean round = (o1 == '1');
 boolean accumulate = (o0 == '1');
```

**Assembler Symbols**

- **<V>** Is a width specifier, encoded in “immh”:
  
<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<d>** Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;shift&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;shift&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 0 1 0 1 1 1 1 1 1 0 | != 0000 | immh | 0 0 1 1 0 1 | Rn | Rd |

SRSRA &lt;V&gt;&lt;d&gt;, &lt;V&gt;&lt;n&gt;, #&lt;shift&gt;

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh&lt;3&gt; != '1' then UNDEFINED;
integer esize = 8 &lt;&lt; 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 |
|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 0Q 0 0 1 1 1 1 1 0 | != 0000 | immh | 0 0 1 1 0 1 | Rn | Rd |

SRSRA &lt;Vd&gt;.&lt;T&gt;, &lt;Vn&gt;.&lt;T&gt;, #&lt;shift&gt;

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh&lt;3&gt;:Q == '10' then UNDEFINED;
integer esize = 8 &lt;&lt; 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

&lt;V&gt; 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>

&lt;d&gt; 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>25</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

### Operation

**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;
**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
```

**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 1 1 1 0 | size | 1 | Rm         | 0 1 0 0 0 1 | Rn | Rd
 U  R  S
```

**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>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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**.

![Hexadecimal notation](image)

```
SSHLL\{2\} <Vd>..<Ta>, <Vn>..<Tb>, #<shift>
```

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>01xx</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</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 “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>

**<shift>** Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```
### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTL, SXTL2</td>
<td>immh == '000' &amp; BitCount(immh) == 1</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and 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 0 0 1 | Rd |
| U | immh | o1 o0 |
```

SSH <V<d>, <V<n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> !='1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 1 0 1 1 1 1 1 0 | != 0000 | immh | 0 0 0 0 0 1 | Rd |
| U | immh | o1 o0 |
```

SSH <Vd>, <T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>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>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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 |
|-----------------|-----------------|--------------|-------------------|
| U immh 01 00    |

SSRA \(<V><d>, <V><n>, #<shift>

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{if } \text{immh} &< 3 > \neq '1' \text{ then UNDEFINED;} \\
\text{integer } \text{esize} &= 8 << 3; \\
\text{integer } \text{datasize} &= \text{esize}; \\
\text{integer } \text{elements} &= 1; \\
\text{integer } \text{shift} &= (\text{esize} \times 2) - \text{UInt}(\text{immh} : \text{immb}); \\
\text{boolean } \text{unsigned} &= (U == '1'); \\
\text{boolean } \text{round} &= (o1 == '1'); \\
\text{boolean } \text{accumulate} &= (o0 == '1');
\end{align*}
\]

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|--------------|-------------------|
| U immh 01 00    |

SSRA \(<Vd>.,<T>,<Vn>.,<T>, #<shift>

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{if } \text{immh} == '0000' \text{ then SEE(asimdimm);} \\
\text{if } \text{immh} < 3 : Q == '10' \text{ then UNDEFINED;} \\
\text{integer } \text{esize} &= 8 << \text{HighestSetBit}(\text{immh}); \\
\text{integer } \text{datasize} &= \text{if } Q == '1' \text{ then } 128 \text{ else } 64; \\
\text{integer } \text{elements} &= \text{datasize} \text{ DIV } \text{esize}; \\
\text{integer } \text{shift} &= (\text{esize} \times 2) - \text{UInt}(\text{immh} : \text{immb}); \\
\text{boolean } \text{unsigned} &= (U == '1'); \\
\text{boolean } \text{round} &= (o1 == '1'); \\
\text{boolean } \text{accumulate} &= (o0 == '1');
\end{align*}
\]

Assembler Symbols

| <V>      | Is a width specifier, encoded in “immh”: |
|-----------------|-----------------|--------------|-------------------|
| immh 0xxx   | RESERVED        |
| immh 1xxx   | 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 name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_Advanced_SIMD_modifiedImmediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modifiedImmediate</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();
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.
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 CPACR_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 | Rn | Rd | 0 | 1 |
```

SSUBL2<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

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register, while the SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | Q | 0 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rd | 0

SSUBW(2) <Vd>,<Ta>, <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 unsigned = (U == '1');

Assembler Symbols

2 
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
Q   2
0   [absent]
1   [present]
```

<Vd> 
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> 
Is an arrangement specifier, encoded in "size":

```
size   <Ta>
00   8H
01   4S
10   2D
11   RESERVED
```

<Vn> 
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> 
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> 
Is an arrangement specifier, encoded in "size:Q":

```
size   Q   <Tb>
00   0   8B
00   1   16B
01   0   4H
01   1   8H
10   0   2S
10   1   4S
11   x   RESERVED
```
Operation

```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.
ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|
| 0   | Q 0 0 1 1 0 0 0 0 0 0 0 0 0 0 x 1 x | size | Rn | Rt |
| L   | opcode                        |
```

One register (opcode == 0111)

ST1 { <Vt>.<T> }, [<Xn|SP>]

Two registers (opcode == 1010)

ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

Three registers (opcode == 0110)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

Four registers (opcode == 0010)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|
| 0   | Q 0 0 1 1 0 0 1 0 0 0 | Rm | x | x | 1 | x | size | Rn | Rt |
| L   | opcode                        |
```
One register, immediate offset (Rm == 11111 && opcode == 0111)

ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>

One register, register offset (Rm != 11111 && opcode == 0111)

ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Two registers, register offset (Rm != 11111 && opcode == 1010)

ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

Three registers, register offset (Rm != 11111 && opcode == 0110)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Four registers, register offset (Rm != 11111 && opcode == 0010)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in “Q”: 
For the two registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#8</td>
</tr>
<tr>
<td>1</td>
<td>#16</td>
</tr>
</tbody>
</table>

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

\[
\begin{align*}
\text{MemOp memop} &= \text{if } L == '1' \text{ then MemOp LOAD else MemOp STORE;} \\
\text{integer datasize} &= \text{if } Q == '1' \text{ then 128 else 64;} \\
\text{integer esize} &= 8 \ll \text{UInt(size);} \\
\text{integer elements} &= \text{datasize DIV esize;} \\
\text{integer rpt;} & \quad \text{// number of iterations} \\
\text{integer selem;} & \quad \text{// structure elements} \\
\text{case opcode of} \\
\quad \text{when '0000' rpt = 1; selem = 4; } \quad \text{// LD/ST4 (4 registers)} \\
\quad \text{when '0010' rpt = 4; selem = 1; } \quad \text{// LD/ST1 (4 registers)} \\
\quad \text{when '0100' rpt = 1; selem = 3; } \quad \text{// LD/ST3 (3 registers)} \\
\quad \text{when '0110' rpt = 3; selem = 1; } \quad \text{// LD/ST1 (3 registers)} \\
\quad \text{when '0111' rpt = 1; selem = 1; } \quad \text{// LD/ST1 (1 register)} \\
\quad \text{when '1000' rpt = 1; selem = 2; } \quad \text{// LD/ST2 (2 registers)} \\
\quad \text{when '1010' rpt = 2; selem = 1; } \quad \text{// LD/ST1 (2 registers)} \\
\quad \text{otherwise UNDEFINED;} \\
\end{align*}
\]

// .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
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

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>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
8-bit, immediate offset \( (Rm == 11111 && opcode == 000) \)

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**

\(<Vt>\) \hspace{1em} Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\(<\text{index}>\) \hspace{1em} 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".

\(<\text{Xn}|\text{SP}>\) \hspace{1em} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{Xm}>\) \hspace{1em} Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
        index = UInt(Q:S);  // S[0-3]
    else
        if S == '1' then UNDEFINED;
        index = UInt(Q);  // D[0-1]
        scale = 3;
    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_Store;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
offs = Zeros();
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register
      Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
      V[t] = rval;
    else // memop == MemOp_STORE
      // extract from one lane of 128-bit register
      Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | size | Rn | Rt |
|-----------------|-----------------|-----------------|
| L               | opcode          |

ST2 { <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | size | Rm | 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>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Xn|SP> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<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>

<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;
```

ST2 (multiple structures)
Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
    if wback then
        if m != 31 then
            offs = X[m];
        if n == 31 then
            SP[] = address + offs;
        else
            X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**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 | O | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |

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
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 | Rm | x | x | 0 | S | size | Rn | Rt |

L R    opcode
8-bit, immediate offset (Rm == 11111 && opcode == 000)

ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2

8-bit, register offset (Rm != 11111 && opcode == 000)

ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = $\textbf{UInt}$ (opcode<2:1>);
integer scale = init_scale;
integer selem = $\textbf{UInt}$ (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 = $\textbf{UInt}$ (size);
    replicate = TRUE;
  when 0
    index = $\textbf{UInt}$ (Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = $\textbf{UInt}$ (Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = $\textbf{UInt}$ (Q:S);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = $\textbf{UInt}$ (Q);  // D[0-1]
    scale = 3;

$\textbf{MemOp}$ memop = if L == '1' then $\textbf{MemOp\ LOAD}$ else $\textbf{MemOp\ STORE}$;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**ST3 (multiple structures)**

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

### No offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

L  

**opcode**

```plaintext
ST3 {<Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<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 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

L  

**opcode**

#### Immediate offset (Rm == 11111)

```plaintext
ST3 {<Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

#### Register offset (Rm != 11111)

```plaintext
ST3 {<Vt>.<T>, <Vt2>.<T>, <Vt3>.<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>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- `<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
    if wback then
        if m != 31 then
            offs = X[m];
        if n == 31 then
            SP[] = address + offs;
        else
            X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**ST3 (single structure)**

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Q   | 0   | 0   | 1   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | x   | x   | 1   | S   | size | Rn  | Rt  |
| L   | R   | opcode |

#### 8-bit (opcode == 001)


#### 16-bit (opcode == 011 && size == x0)

ST3 `{ <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]`

#### 32-bit (opcode == 101 && size == 00)

ST3 `{ <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]`

#### 64-bit (opcode == 101 && S == 0 && size == 01)

ST3 `{ <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]`

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Q   | 0   | 0   | 1   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | x   | x   | 1   | S   | size | Rn  | Rt  |
| L   | R   | opcode |

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
8-bit, immediate offset (Rm == 11111 && opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3

8-bit, register offset (Rm != 11111 && opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3 // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
    scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bear(64) offs;
bear(128) rval;
bear(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize); 
    offs = offs + ebytes;
    t = (t + 1) MOD 32; 
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 Q 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 E | size Rn Rt |
| opcode         |


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 Q 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 E | size Rn Rt |
| opcode         |

Immediate offset (Rm == 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;  // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;  // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;  // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;  // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;  // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 3;  // 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;
```

ST4 (multiple structures)
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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

8-bit (opcode == 001)

```
```

16-bit (opcode == 011 && size == x0)

```
```

32-bit (opcode == 101 && size == 00)

```
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
```

integer t = UInt(Rt);
integer n =UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

```
```

integer t = UInt(Rt);
integer n =UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
8-bit, immediate offset (Rm == 11111 && opcode == 001)

8-bit, register offset (Rm != 11111 && opcode == 001)

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3 // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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.

32-bit (opc == 00)

STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]

64-bit (opc == 01)

STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]  // Empty.

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
def integer n = UInt(Rn);
def integer t = UInt(Rt);
def integer t2 = UInt(Rt2);
def if opc == '11' then UNDEFINED;
def integer scale = 2 + UInt(opc);
def integer datasize = 8 << scale;
def bits(64) offset = LSL(SignExtend(imm7, 64), scale);
def boolean tag_checked = n != 31;
```
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VECSTREAM] = data1;
Mem[address+dbytes, dbytes, AccType_VECSTREAM] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| opc             | imm7            | Rt2             | Rn              | Rt              |

32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| opc             | imm7            | Rt2             | Rn              | Rt              |

32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>!

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>!

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>!

boolean wback = TRUE;
boolean postindex = FALSE;

Signed offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| opc             | imm7            | Rt2             | Rn              | Rt              |
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;
bis(64) 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
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VEC] = data1;
Mem[address+dbytes, dbytes, AccType_VEC] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The
address that is used for the store is calculated from a base register value and an immediate offset.
Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rt</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>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 00)

STR <Bt>, [<Xn|SP>], #<simm>

16-bit (size == 01 && opc == 00)

STR <Ht>, [<Xn|SP>], #<simm>

32-bit (size == 10 && opc == 00)

STR <St>, [<Xn|SP>], #<simm>

64-bit (size == 11 && opc == 00)

STR <Dt>, [<Xn|SP>], #<simm>

128-bit (size == 00 && opc == 10)

STR <Qt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
binary(64) offset = SignExtend(imm9, 64);

Pre-index

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm9</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>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rt</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>
</tr>
</tbody>
</table>
8-bit (size == 00 && opc == 00)

STR <Bt>, [<Xn|SP>, #<simm>]

16-bit (size == 01 && opc == 00)

STR <Ht>, [<Xn|SP>, #<simm>]

32-bit (size == 10 && opc == 00)

STR <St>, [<Xn|SP>, #<simm>]

64-bit (size == 11 && opc == 00)

STR <Dt>, [<Xn|SP>, #<simm>]

128-bit (size == 00 && opc == 10)

STR <Qt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 1  | 1  | 1  | 0  | 1  | x  | 0  | imm12 | |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

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;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
**Assembler Symbols**

<i>Bt</i>  
Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<i Dt</i>  
Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<i>Ht</i>  
Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<i>Qt</i>  
Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<i>St</i>  
Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<i>Xn|SP</i>  
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<i>simm</i>  
Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<i>pimm</i>  
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 <i>pimm</i>/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 <i>pimm</i>/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 <i>pimm</i>/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 <i>pimm</i>/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
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STR (register, SIMD&FP)**

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>S</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
</tr>
</tbody>
</table>

8-fsreg, STR-8-fsreg (size == 00 & opc == 00 & option != 011)

STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]  

8-fsreg, STR-8-fsreg (size == 00 & opc == 00 & option == 011)

STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  

16-fsreg, STR-16-fsreg (size == 01 & opc == 00)

STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]  

32-fsreg, STR-32-fsreg (size == 10 & opc == 00)

STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]  

64-fsreg, STR-64-fsreg (size == 11 & opc == 00)

STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]  

128-fsreg, STR-128-fsreg (size == 00 & opc == 10)

STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]  

integer scale = UInt(opc<1>::size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED; /* sub-word index
ExtendTime 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, extend\_type, shift);
if \texttt{HaveMTEExt}() then
    \texttt{SetTagCheckedInstruction}(tag\_checked);
\texttt{CheckFPAadvSIMDEnabled64}();
bits(64) address;
bits(datasize) data;
if \texttt{n == 31} then
    \texttt{CheckSPAlignment}();
    address = \texttt{SP}[n];
else
    address = \texttt{X}[n];
address = address + offset;
case memop of
    when \texttt{MemOp\_STORE}
        data = \texttt{V}[t];
        \texttt{Mem}[address, datasize DIV 8, AccType\_VEC] = data;
    when \texttt{MemOp\_LOAD}
        data = \texttt{Mem}[address, datasize DIV 8, AccType\_VEC];
        \texttt{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</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>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 00)

STUR <Bt>, [<Xn|SP>{, #<simm>}

16-bit (size == 01 && opc == 00)

STUR <Ht>, [<Xn|SP>{, #<simm>}

32-bit (size == 10 && opc == 00)

STUR <St>, [<Xn|SP>{, #<simm>}

64-bit (size == 11 && opc == 00)

STUR <Dt>, [<Xn|SP>{, #<simm>}

128-bit (size == 00 && opc == 10)

STUR <Qt>, [<Xn|SP>{, #<simm>}

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bites(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
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**SUB (vector)**

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>1</td>
<td>0 0 0 0 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

\[
\text{SUB } <V><d>, <V><n>, <V><m>}
\]

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Rm) \);
if size != '11' then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (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>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\text{SUB } <Vd><T>, <Vn><T>, <Vm><T>}
\]

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Rm) \);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
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**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bite eso) element1;
bite eso) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, eso];
    element2 = Elem[operand2, e, eso];
    if sub_op then
        Elem[result, e, eso] = element1 - element2;
    else
        Elem[result, e, eso] = 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.

![Register Format](image)

**Asmcer 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>
<th>1</th>
<th>[present]</th>
</tr>
</thead>
</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

```plaintext
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;
for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SUDOT (by element)

Dot product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2 to Armv8.5, this is an optional instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. \textit{ID\_AA64ISAR1\_EL1}.I8MM indicates whether this instruction is supported.

Vector
(Armv8.6)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | L  | M  | Rm | 1  | 1  | 1  | 1  | H  | 0  | Rn | Rd |

US

SUDOT \(<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]\)

```java
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.
- \(<\text{index}>\) Is the immediate index of a quadtuplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.
Operation

\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(\text{datasize}) \text{operand1} = \mathbf{V}[n];}
\texttt{bits(128) \text{operand2} = \mathbf{V}[m];}
\texttt{bits(\text{datasize}) \text{operand3} = \mathbf{V}[d];}
\texttt{bits(\text{datasize}) \text{result};}

\texttt{for e = 0 to elements-1}
\quad \texttt{bits(32) \text{res} = \texttt{Elem}[\text{operand3}, e, 32];}
\quad \texttt{for b = 0 to 3}
\quad \quad \texttt{integer \text{element1} = \texttt{Int}(\texttt{Elem}[\text{operand1}, 4*\text{e+b}, 8], \text{op1}\_\text{unsigned});}
\quad \quad \texttt{integer \text{element2} = \texttt{Int}(\texttt{Elem}[\text{operand2}, 4*\text{i+b}, 8], \text{op2}\_\text{unsigned});}
\quad \quad \texttt{\text{res} = \text{res} + \text{element1} \times \text{element2};}
\quad \texttt{\texttt{Elem}[\text{result}, e, 32] = \text{res};}
\texttt{\mathbf{V}[d] = \text{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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rn | Rd |

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

| 31 30 29 28 27 26 25 24 23 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 | 0 | 0 | 1 | 1 | 1 | 0 | Rd |

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.

<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>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>

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
biasT(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
inTEGRal op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SXTL, SXTL2

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values. The SXTL instruction extracts the source vector from the lower half of the source register, while the SXTL2 instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of SSSHLL, SSSHLL2. This means:

- The encodings in this description are named to match the encodings of SSSHLL, SSSHLL2.
- The description of SSSHLL, SSSHLL2 gives the operational pseudocode for this instruction.

$$\begin{array}{cccccccccccc}
\hline
& Q & 0 & 1 & 1 & 1 & 1 & 0 & != 0000 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & Rn & Rd & U & immh & immb
\end{array}$$

SXTL{2} <Vd>.<Ta>, <Vn>.<Tb>

is equivalent to

SSHL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when BitCount(immh) == 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":

$$\begin{array}{c|c}
\text{Q} & \text{2} \\
\hline
0 & [\text{absent}] \\
1 & [\text{present}] \\
\end{array}$$

<Vd>  

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  

Is an arrangement specifier, encoded in "immh":

$$\begin{array}{c|c}
\text{immh} & \text{<Ta>} \\
\hline
0000 & \text{SEE Advanced SIMD modified immediate} \\
0001 & 8H \\
001x & 4S \\
01xx & 2D \\
1xxx & \text{RESERVED} \\
\end{array}$$

<Vn>  

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb>  

Is an arrangement specifier, encoded in “immh:Q”:

$$\begin{array}{c|c|c|c}
\text{immh} & \text{Q} & \text{<Tb>} \\
\hline
0000 & x & \text{SEE Advanced SIMD modified immediate} \\
0001 & 0 & 8B \\
0001 & 1 & 16B \\
001x & 0 & 4H \\
001x & 1 & 8H \\
01xx & 0 & 2S \\
01xx & 1 & 4S \\
1xxx & x & \text{RESERVED} \\
\end{array}$$

Operation

The description of SSSHLL, SSSHLL2 gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | Rm | 0  | len | 0  | 0  | 0  | 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
d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

datasize = if Q == '1' then 128 else 64;
elements = datasize DIV 8;
regs = UInt(len) + 1;
is_tbl = (op == '0');
```

### Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>` is an arrangement specifier, encoded in "Q":
  
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>` for the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.
  
  For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

- `<Vn+1>` is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.
- `<Vn+2>` is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.
- `<Vn+3>` is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.
- `<Vm>` is the name of the SIMD&FP index register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Two register table (len == 01)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>

Three register table (len == 10)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>

Four register table (len == 11)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>

Single register table (len == 00)

TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.
For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.
<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.
<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.
<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.
Operation

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.
**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](image)

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```plaintext
Assemblers Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
```
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

By using this instruction with TRN1, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```plaintext
TRN2 <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;

for p = 0 to pairs-1
   Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
   Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<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>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>U</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
  element1 = Int(Elem(operand1, e, esize), unsigned);
  element2 = Int(Elem(operand2, e, esize), unsigned);
  absdiff = Abs(element1-element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register, while the UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Hexadecimal representation of UABAL, UABAL2 instruction](image)

```plaintext
UABAL{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":

```
Q 2
0 [absent]
1 [present]
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

```
size <Ta>
00 8H
01 4S
10 2D
11 RESERVED
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

```
size Q <Tb>
00 0 8B
00 1 16B
01 0 4H
01 1 8H
10 0 2S
10 1 4S
11 x RESERVED
```

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
number element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  absdiff = Abs(element1-element2)<2*esize-1:0>;
  Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Asmmbler 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.

```
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>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in &quot;Q&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ta>** Is an arrangement specifier, encoded in "size":

<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();
bots(datasize) operand1 = Vpart[n, part];
bots(datasize) operand2 = Vpart[m, part];
bots(2*datasize) result;
integer element1;
integer element2;
bots(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.
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.

$integer\ d = UInt(Rd);$  
$integer\ n = UInt(Rn);$  
$\text{if } size == '11'\ \text{then } UNDEFINED;$  
$integer\ esize = 8 \ll\ UInt(size);$  
$integer\ datasize = if\ Q == '1'\ \text{then } 128\ \text{else } 64;$  
$integer\ elements = datasize\ \text{DIV}\ (2 \times esize);$  
$boolean\ acc = (op == '1');$  
$boolean\ unsigned = (U == '1');$  

**Assembler Symbols**

<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>

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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.

```
0 | Q | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 0 | 0 | 0 | 0 | Rn | 0 | Rd | 0 | 1

UADDL2 {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 datasize = 64;
integer part = Uint(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in &quot;Q&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\[
\begin{array}{cccccccccccccccc}
\hline
\text{U} & 0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & \text{size} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \text{Rn} & \text{Rd} & \text{op}
\end{array}
\]

UADDLP <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.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<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>

<Ta> 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;
if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

- **<V>** Is the destination width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<d>** Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the “Rn” field.

- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
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 | 0 | size | 1 | Rm | 0 0 | 0 | 1 | 0 | 0 | Rn | Rd |
|-----------------------------------------------|----------------|
| UADDW2  <Vd>, <Ta>, <Vn>, <Vm>, <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>
</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>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.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | immh|

UCVTF `<V><d>, <V><cn>, #<fbits>`

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then 
  if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
  integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
  integer data = data = esize;
  integer elements = 1;

integer fr = (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  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | immh |

UCVTF `<Vd>, <T>, <Vn>, <T>, #<fbits>`

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then 
  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 data = if Q == '1' then 128 else 64;
  integer elements = data DIV esize;

integer fr = (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>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_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bites(128) result = if merge then V[d] else Zeros();

for e = 0 to elements - 1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding);

V[d] = result;
UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|
| 0 1 1 1 1 1 0 0 | 1 1 1 1 0 0 | 1 1 1 0 1 1 0 | U |
| Rn | Rd |

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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|
| 0 1 1 1 1 1 0 0 | sz | 1 0 | 0 | 0 | 0 | 1 1 1 0 1 1 0 | Rn | Rd |

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)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|:--:|
| 0 | Q | 1 0 | 1 1 1 0 0 | 1 1 1 0 1 1 0 | Rn | Rd |

UCVTF (vector, integer)
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |
| U |

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

FPCRTypetype fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

FPRounding rounding = FPRoundingMode(fpcr);
bits(esimal) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.
32-bit to half-precision (sf == 0 & ftype == 11)
(Armv8.2)
UCVTF <Hd>, <Wn>, #<fbits>

32-bit to single-precision (sf == 0 & ftype == 00)
UCVTF <Sd>, <Wn>, #<fbits>

32-bit to double-precision (sf == 0 & ftype == 01)
UCVTF <Dd>, <Wn>, #<fbits>

64-bit to half-precision (sf == 1 & ftype == 11)
(Armv8.2)
UCVTF <Hd>, <Xn>, #<fbits>

64-bit to single-precision (sf == 1 & ftype == 00)
UCVTF <Sd>, <Xn>, #<fbits>

64-bit to double-precision (sf == 1 & ftype == 01)
UCVTF <Dd>, <Xn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  end;
if sf == '0' & scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits> For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".

UCVTF (scalar, fixed-point)
For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```c
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, TRUE, fpcr, rounding);
V[d] = fltval;
```
UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the
general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and
writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception
results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see
Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| sf | 0 | 0 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
| rmode | opcode |
32-bit to half-precision (sf == 0 & ftype == 11)  
(Armv8.2)

UCVT <Hd>, <Wn>

32-bit to single-precision (sf == 0 & ftype == 00)

UCVT <Sd>, <Wn>

32-bit to double-precision (sf == 0 & ftype == 01)

UCVT <Dd>, <Wn>

64-bit to half-precision (sf == 1 & ftype == 11)  
(Armv8.2)

UCVT <Hd>, <Xn>

64-bit to single-precision (sf == 1 & ftype == 00)

UCVT <Sd>, <Xn>

64-bit to double-precision (sf == 1 & ftype == 01)

UCVT <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

```
CheckFPAdvSIMEnabled64();

FPCRType  fpcr = FPCR[];
boolean  merge = IsMerging(fpcr);
integer  fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem(fltval, 0, fltsize) = FixedToFP(intval, 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)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    0 | Q | 1 | 0 | 1 | 1 | 1 | size | L | M | Rm | 1 | 1 | 0 | H | 0 | Rn | Rd |
    U
```

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>88</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<index> Is the element index, encoded in the "H:L" fields.
Operation

CheckFPAdvSIMDEnabled64();
b_bits(datasize) operand1 = V[n];
b_bits(128) operand2 = V[m];
b_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_AAA41ISAR0_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

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>0</th>
<th>Rm</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
</tr>
</tbody>
</table>

UDOT <Vd>,<Ta>, <Vn>,<Tb>, <Vm>,<Tb>

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
  integer res = 0;
  integer element1, element2;
  for i = 0 to 3
    if signed then
      element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
    else
      element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

**UHADD** <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);
type m = UInt(Rm);
if size == '11' then UNDEFINED;
type esize = 8 << UInt(size);
type datasize = if Q == '1' then 128 else 64;
type 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**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
type element1;
type element2;
type 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 response of this instruction to 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.

| 31 30 29 28 27 26 25 24 23 22 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 | size | 1 | Rm | 0 0 1 0 0 1 | Rn | Rd |

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":

| size | Q | <T> |
| 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.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
     ○ The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the $\text{CPACR_EL1}$, $\text{CPTR_EL2}$, and $\text{CPTR_EL3}$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 | 1  | Rm  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | Rd  | 0  | 1  |

**UMAX $<Vd>..<T>$, $<Vn>..<T>$, $<Vm>..<T>$**

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
integer $m = \text{UInt}(Rm)$;
if $\text{size} == '11'$ then UNDEFINED;
integer $\text{esize} = 8 << \text{UInt}(\text{size})$;
integer $\text{datasize} = \text{if} \ Q == '1' \ \text{then} \ 128 \ \text{else} \ 64$;
integer $\text{elements} = \text{datasize} \text{DIV} \text{esize}$;

boolean $\text{unsigned} = (U == '1')$;
boolean $\text{minimum} = (o1 == '1')$;

**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>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**$<Vn>$**
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**$<Vm>$**
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer 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 $\text{PSTATE.DIT}$ 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.
UMAXP

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

- **Assembler Symbols**
  - `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
  - `<T>` is an arrangement specifier, encoded in "size:Q":
    - size | Q | <T>
      - 00 | 0 | 8B
      - 00 | 1 | 16B
      - 01 | 0 | 4H
      - 01 | 1 | 8H
      - 10 | 0 | 2S
      - 10 | 1 | 4S
      - 11 | x | RESERVED
  - `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
  - `<Vm>` is the name of the second SIMD&FP source register, encoded in 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.
**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.

```
+---+---+---+---+---+---+---+---+---+---+---+---+
| 31| 30| 29| 28| 27| 26| 25| 24| 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| 00|  0|  0|
|    |    |    |    |    |    |    |    |    |    |    |    |
+---+---+---+---+---+---+---+---+---+---+---+---+
|     size | 00 | 8B |
|    Q     |  0 |  1 |
|    Rd    | 10 |  0 |
|         | 11 | x  |
```

### Assembler Symbols

- `<V>` is the destination widthspecifier, 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
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');
```

Operation

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean min = (op == '1');
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean min = (op == '1');
```

```plaintext
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.

<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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

```python
def 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 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.

```
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.

```
| 31 30 29 28 27 26 25 24 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                | size             | op               |
```

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>&lt;d&gt;, &lt;Vn&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>&lt;d&gt;</th>
<th>Is the number of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&lt;Vn&gt;</th>
<th>Is the name of the SIMD&amp;FP source register, encoded in the “Rn” field.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
<th>Is an arrangement specifier, encoded in “size:Q”:</th>
</tr>
</thead>
</table>
| 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>;
```

UMINV
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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.

```
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

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>absent</td>
</tr>
<tr>
<td>1</td>
<td>present</td>
</tr>
</tbody>
</table>
```

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```
Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements - 1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<<2*esize-1:0>
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 0 | 0 | Rn | Rd | 01

UMLAL2 {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>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) 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.

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

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMLSL instruction extracts each source vector from the lower half of each source register, while the UMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | Rn |    |    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |

UMLSL2 \( \{2\} \): \(<Vd>\), \(<Ta>\), \(<Vn>\), \(<Vm>\), \(<Vm>\)

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

```
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.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an optional instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.18MM indicates whether this instruction is supported.

Vector
(Armv8.6)

![Matrix Multiplication Diagram]

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);
**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 `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias **MOV (to general)**.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm5</th>
<th></th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**32-bit (Q == 0)**

UMOV `<Wd>, <Vn>.<Ts>[<index>]`

**64-reg,UMOV-64-reg (Q == 1 && imm5 == x1000)**

UMOV `<Xd>, <Vn>.<Ts>[<index>]`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
  when '0xxxx1' size = 0; // UMOV Wd, Vn.B
  when '0xxx10' size = 1; // UMOV Wd, Vn.H
  when '0xx100' size = 2; // UMOV Wd, Vn.S
  when '1x1000' size = 3; // UMOV Xd, Vn.D
  otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Ts>` For the 32-bit variant: is an element size specifier, encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
</tbody>
</table>

For the 64-reg,UMOV-64-reg variant: is an element size specifier, encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xx100</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<index>` For the 32-bit variant: is the element index encoded in "imm5":

---

**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 `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias **MOV (to general)**.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm5</th>
<th></th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**32-bit (Q == 0)**

UMOV `<Wd>, <Vn>.<Ts>[<index>]`

**64-reg,UMOV-64-reg (Q == 1 && imm5 == x1000)**

UMOV `<Xd>, <Vn>.<Ts>[<index>]`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
```
### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to general)</td>
<td>imm5 == 'x100'</td>
</tr>
<tr>
<td>MOV (to general)</td>
<td>imm5 == 'xx100'</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxsize) 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.

---

For the 64-reg, UMOV-64-reg variant: is the element index encoded in "imm5<4>".
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></td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UMULL2 \(<Vd>, <Vn>, <Vm>, <Ts>[<index>]\)

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;

\begin{align*}
\text{case size of} & \\
\text{when '01'} & \text{index} = \text{UInt}(H:L:M); Rmhi = '0'; \\
\text{when '10'} & \text{index} = \text{UInt}(H:L); Rmhi = M; \\
\text{otherwise} & \text{UNDEFINED};
\end{align*}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

- 2: Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

- <Vd>: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Ta>: Is an arrangement specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- <Vn>: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Tb>: Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- <Vm>: Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

UMULL, UMULL2 (by element)
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**

```plaintext
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.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | Qi | 0 1 1 1 0 | size | 1 | Rm | 1 1 0 0 0 0 | Rn | Rd |

UMULL2{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 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.
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>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>
```

UQADD <V><d>, <V><n>, <V><m>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>
```

UQADD <Vd>..<T>, <Vn>..<T>, <Vm>..<T>

```java
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.

<T>    Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<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{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;
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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 **UQSHL**.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 1  | 0  | 1  | 1  | Rm | 0  | 1  | 0  | 1  | 1  | 1  | Rn | Rd |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U  | R  | S |

**UQRSHL** `<V><d>`, `<V><n>`, `<V><m>`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 1  | 0  | 1  | 1  | Rm | 0  | 1  | 0  | 1  | 1  | 1  | Rn | Rd |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U  | R  | S |

**UQRSHL** `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<T>`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

### Assembler Symbols

**<V>**

Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**<d>**

Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);  // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
        if saturating then
            (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
            if sat then FPSR.QC = '1';
        else
            Elem[result, e, esize] = element<esize-1:0>;
    end
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 UQRSHRN.

The UQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 != '0000' immb 1 0 0 1 1 Rn Rd
```

```
UQRSHRN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immb == '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 1 0 1 1 1 1 0 != '0000' immb 1 0 0 1 1 Rn Rd
```

```
UQRSHRN{2} <Vd>..<Tb>, <Vn>..<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immb == '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":

| Q |  
|---|---|
| 0 | [absent] |
| 1 | [present] |

<\(V_d\)>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\(T_b\)>  Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>(&lt;T_b&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>0011</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>0010</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>

<\(V_n\)>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<\(T_a\)>  Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;T_a&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>0011</td>
<td>4S</td>
</tr>
<tr>
<td>0110</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<\(V_b\)>  Is the destination width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;V_b&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.

<\(V_a\)>  Is the source width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;V_a&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<\(n\)>  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\(\text{shift}\)>  For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immh: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>(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:immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;\text{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  | 1  | 0  | != | 0000 | immh | 0  | 1  | 1  | 1  | 0  | 1  | Rn  |     |     | Rd  |     |     |     |     |     |     |     |
| U  | immh | op |

UQSHL <V><d>, <V><cn>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

**Vector**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | != | 0000 | immh | 0  | 1  | 1  | 1  | 0  | 1  | Rn  |     |     | Rd  |     |     |     |     |     |     |     |
| U  | immh | op |

UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SFE(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>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0 8B</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------|--------------------------------------|--------------------------------------|
| 0 1 1 1 1 1 1 0 | size | 1 | Rm | 0 1 0 | 0 1 1 | Rn | Rd |
| U | R | S |

UQSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' & size != '11' then UNDEFINED;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------|--------------------------------------|--------------------------------------|
| 0 1 0 1 1 1 0 | size | 1 | Rm | 0 1 0 | 0 1 1 | Rn | Rd |
| U | R | S |

UQSHL <Vd><T>, <Vn><T>, <Vm><T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 B</td>
<td></td>
</tr>
<tr>
<td>01 H</td>
<td></td>
</tr>
<tr>
<td>10 S</td>
<td></td>
</tr>
<tr>
<td>11 D</td>
<td></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 Q</th>
<th>&lt;T&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 0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11 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>;
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | != 0000 | immb | 1  | 0  | 0  | 1  | 0  | 1  |     | Rn  |     | Rd  |
| U  | immh | op |
```

UQSHRN <Vb><d>, <Va><n>, #<shift>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | != 0000 | immb | 1  | 0  | 0  | 1  | 0  | 1  |     | Rn  |     | Rd  |
| U  | immh | op |
```

UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SFE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```
### Assembler Symbols

2. Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

\(<Tb>\) Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>( \text{immh} )</th>
<th>( Q )</th>
<th>( \text{(&lt;Tb&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>( x )</td>
<td>\text{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>( \text{immh} )</th>
<th>( \text{(&lt;Ta&gt;)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\text{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>\text{RESERVED}</td>
</tr>
</tbody>
</table>

\(<Vb>\) Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>( \text{immh} )</th>
<th>( \text{(&lt;Vb&gt;)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\text{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>\text{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>( \text{immh} )</th>
<th>( \text{(&lt;Va&gt;)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\text{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>\text{RESERVED}</td>
</tr>
</tbody>
</table>

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

\(<\text{shift}>\) 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>( \text{immh} )</th>
<th>( \text{(&lt;\text{shift}&gt;)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\text{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>\text{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>( \text{immh} )</th>
<th>( \text{(&lt;\text{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-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>\text{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;
**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

<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>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U

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

<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>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U

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.
- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "size:Q".
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  diff = element1 - element2;
  (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
  if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQXTN, UQXTN2**

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit \(\text{FPSR}.\text{QC}\) is set.

The \(\text{UQXTN}\) instruction writes the vector to the lower half of the destination register and clears the upper half, while the \(\text{UQXTN2}\) 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 \(\text{CPACR}_\text{EL1}, \text{CPTR}_\text{EL2},\) and \(\text{CPTR}_\text{EL3}\) 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

![](image)

\[
\begin{array}{ccccccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 0 & \text{size} & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \text{Rn} & \text{Rd} \\
\end{array}
\]

\(\text{UQXTN} \ <\text{Vb}\>\ <\text{d}>\), \(<\text{Va}\>\ <\text{n}>\)

\[
\begin{align*}
\text{integer } d & = \text{UInt}(\text{Rd}); \\
\text{integer } n & = \text{UInt}(\text{Rn}); \\
\text{if size == '11' then UNDEFINED;} \\
\text{integer esize} & = 8 \ll \text{UInt}(\text{size}); \\
\text{integer datasize} & = \text{esize}; \\
\text{integer part} & = 0; \\
\text{integer elements} & = 1; \\
\text{boolean unsigned} & = (\text{U} == '1');
\end{align*}
\]

### Vector

![](image)

\[
\begin{array}{ccccccccccccc}
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & \text{size} & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \text{Rn} & \text{Rd} \\
\end{array}
\]

\(\text{UQXTN}\{2\} \ <\text{Vd}\>\ .\ <\text{Tb}>\), \(<\text{Vn}\>\ <\text{Ta}>\)

\[
\begin{align*}
\text{integer } d & = \text{UInt}(\text{Rd}); \\
\text{integer } n & = \text{UInt}(\text{Rn}); \\
\text{if size == '11' then UNDEFINED;} \\
\text{integer esize} & = 8 \ll \text{UInt}(\text{size}); \\
\text{integer datasize} & = 64; \\
\text{integer part} & = \text{UInt}(\text{Q}); \\
\text{integer elements} & = \text{datasize DIV esize}; \\
\text{boolean unsigned} & = (\text{U} == '1');
\end{align*}
\]

### 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":

\[
\begin{array}{c|c}
\text{Q} & 2 \\
\hline
0 & [\text{absent}] \\
1 & [\text{present}] \\
\end{array}
\]

\(<\text{Vd}>\)  

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Tb}>\)  

Is an arrangement specifier, encoded in "size:Q":

---

UQXTN, UQXTN2  
Page 1396
<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];
  Elemlen(result, e, esize), sat) = SatQ(Int(element, unsigned), esize, unsigned);
  if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URECPE**

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
```

**Assembling 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**

```c
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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URHADD**

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see **UHADD**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The operand is encoded in the **{<Vd>,<T>}, <Vn>,<T>, <Vm>,<T>** fields.

- **<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
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');
```

```plaintext
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;
```

**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".
- **<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.
URSHL

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 1  | 1  | 0 | 1 | 1 | 0 | 1 | Rn | 1 | Rd |

URSHL $<V><d>$, $<V><n>$, $<V><m>$

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  | 0 | Q | 1  | 0  | 1  | 1  | 1  | 1  | 0  | size | 1  | Rm | 0  | 1  | 0 | 1 | 0 | 1 | 0 | Rn | 1 | Rd |

URSHL $<Vd>..<T>$, $<Vn>..<T>$, $<Vm>..<T>$

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>6x</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>

<Fn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem(operand2, e, esize)<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem(operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem(result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem(result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URSHR**

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see **USHR**.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 1 1 1 1 1 1 1 1 0 | != 0000 | immmb | 0 0 1 0 0 1 | Rn | Rd |
| U | immh | o1 o0 |

**URSHR** `<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);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 0 1 1 1 1 1 0 | != 0000 | immmb | 0 0 1 0 0 1 | Rn | Rd |
| U | immh | o1 o0 |

**URSHR** `<V>d.`.<T>, `<V>n`.<T>, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << `HighestSetBit`(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

**Assembler Symbols**

<`V>` Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<`d>` Is the number of the SIMD&FP destination register, in the “Rd” field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0011</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;
```

Copyright © 2010-2020 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 0 0 0 0 1 1 1 0 0 1 0 | Rn | Rd

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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UNSIGNED ROUNDED 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

```
0 1 1 1 1 1 1 0 | != 0000 | immh | 0 0 1 1 0 1 | Rd
```

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 0 | != 0000 | immh | 0 0 1 1 0 1 | Rd
```

Integer d = UInt(Rd);
Integer n = UInt(Rn);
If immh == '0000' then SEE(asimdimm);
If immh<3>:Q == '10' then UNDEFINED;
Integer esize = 8 << HighestSetBit(immh);
Integer datasize = if Q == '1' then 128 else 64;
Integer elements = datasize DIV esize;

Integer shift = (esize * 2) - UInt(immh:immb);
Boolean unsigned = (U == '1');
Boolean round = (o1 == '1');
Boolean accumulate = (o0 == '1');

**Assembler Symbols**

< V > Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>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 right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**USDOT (by element)**

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an **OPTIONAL** instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| US | Q  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | L  | M  | Rm | 1  | 1  | 1  | 1  | H  | 0  | Rn | Rd |

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 quaduplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.
Operation

```c
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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 to Armv8.5, this is an **optional** instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. **ID_AA64ISAR1_EL1.I8MM** indicates whether this instruction is supported.

**Vector (Armv8.6)**

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 1   | 1   | 1   | 0   | 0   | 1   | 0   | 0   | 1   | 1   | 0   | 1   | 0   | 1   | 1   | 1   | 1   | 1   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | 0   | 1   | 0   | 0   | 1   | 0   | 0   | 1   | 0   | 0   | 1   | 0   | 0   | 1   | 0   | 0   | 1   | 0   | 0   |
```

**Rm** | **Rn** | **Rd**

USDOT `<Vd>`, `<Ta>`, `<Vn>`, `<Tb>`, `<Vm>`, `<Tb>`

if !**HaveInt8MatMulExt()** then UNDEFINED;

integer n = **UInt**(Rn);
integer m = **UInt**(Rm);
integer d = **UInt**(Rd);
integer datatype = if Q == '1' then 128 else 64;
integer elements = datatype 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>88</td>
</tr>
<tr>
<td>1</td>
<td>168</td>
</tr>
</tbody>
</table>

- **<Vm>** is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datatype) operand1 = V[n];
bits(datatype) operand2 = V[m];
bits(datatype) operand3 = V[d];
bits(datatype) 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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USHL

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see USHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10</th>
<th>Rd</th>
<th>Rn</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>0 1 0 0 0 1</td>
<td>0 1 0 0 0 1</td>
<td>0 1 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>R</td>
<td>S</td>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 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>Rd</th>
<th>Rn</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0</td>
<td>0 1 0 0 0 1</td>
<td>0 1 0 0 0 1</td>
<td>0 1 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>R</td>
<td>S</td>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
</tr>
<tr>
<td></td>
<td>0x</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>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
    element = Elem[result, e, esize] = SatQ(element, esize, unsigned);
  if saturating then
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
  else
    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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**.

![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**:Q”:

<table>
<thead>
<tr>
<th><strong>immh</strong></th>
<th>&lt;<strong>Ta</strong>&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><strong>immh</strong></th>
<th><strong>Q</strong></th>
<th>&lt;<strong>Tb</strong>&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>

<**shift**> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “**immh**:immb”:

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```
### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UXTL, UXTL2</td>
<td>immh == '000' &amp; BitCount(immh) == 1</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
   element = Int(Elem[operand, e, esize], unsigned) << shift;
   Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USHR

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right
shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination
SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For
rounded results, see USHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 0 1 1 1 1 1 1 1 1 1 0                         | != 0000       |
| U                                             |
| immh                                          |
| 0 0 0 0 0 0 1                                 |
| Rn                                            |
| Rd                                            |
```

USHR <V><d>, <V><n>, #=><shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer en = 8 << 3;
integer datasize = en;
integer elements = 1;

integer shift = (en * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 0 1 0 1 1 1 1 0                             | != 0000       |
| U                                             |
| immh                                          |
| 0 0 0 0 0 0 1                                 |
| Rn                                            |
| Rd                                            |
```

USHR <Vd>,<T>, <Vn>.<T>, #=><shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

```
<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
  element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. \textit{ID\_AA64ISAR1\_EL1}.I8MM indicates whether this instruction is supported.

Vector
(Armv8.6)

\begin{verbatim}
0 1 0 1 1 0 1 0 | Rm 1 0 1 0 1 1 | Rn | Rd

U

B
\end{verbatim}

USMMLA \(<Vd>.4S, <Vn>.16B, <Vm>.16B$

if !\textit{HaveInt8MatMulExt}() then UNDEFINED;
integer n = \textit{UInt}(Rn);
integer m = \textit{UInt}(Rm);
integer d = \textit{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

\begin{verbatim}
\textit{CheckFPAdvSIMDEnabled64}();
bits(128) operand1 = \textit{V}[n];
bits(128) operand2 = \textit{V}[m];
bits(128) addend = \textit{V}[d];
\textit{V}[d] = \textit{MatMulAdd}(addend, operand1, operand2, TRUE, FALSE);
\end{verbatim}
**USQADD**

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|----------------|----------------|
| size                     | Rn              | Rd              |
```

USQADD `<V><d>, <V><n>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
n = UInt(n);

elements = 1;
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 |
|---------------------------|----------------|----------------|
| size                     | Rn              | Rd              |
```

USQADD `<Vd>.<T>, <Vn>.<T>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;

integer esize = 8 << UInt(size);

integer elements = if Q == '1' then 128 else 64;
unsigned = (U == '1');
```

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in "size":
  ```markdown
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
  ```

- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
basic(datasize) operand = V[n];
basic(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see USRRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 1 0 != 0000 immh 0 0 0 1 0 1 Rd
U

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

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 1 0 != 0000 immh 0 0 0 1 0 1 Rd
U

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”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0 88</td>
</tr>
<tr>
<td>0001</td>
<td>1 16B</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
b曝(bits(datasize) operand = V[n];
b曝(bits(datasize) operand2;
b曝(bits(datasize) result;
b曝(integer round_const = if round then (1 << (shift - 1)) else 0;
b曝(integer element;
b曝(operand2 = if accumulate then V[d] else Zeros();
b曝(for e = 0 to elements-1
            element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
b曝            Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
b曝    V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register, while the USUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 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>Q</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>
    U                                 o1
```

USUBL2 (2) <Vd>, <Ta>, <Vn>, <Tb>

```java
integer d = UInt(Rd);
ingen = UInt(Rn);
ingen m = UInt(Rm);
if size == '11' then UNDEFINED;
ingen esize = 8 << UInt(size);
gen integer datasete = 64;
gen integer part = UInt(Q);
gen integer elements = datasete DIV esize;

boolean sub_op = (o1 == '1');
gen 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>2</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

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
USUBW, USUBW2

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source register, while the USUBW2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  |

USUBW2 <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.
**UXTL, UXTL2**

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The UXTL instruction extracts vector elements from the lower half of the source register, while the UXTL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of **USHLL, USHLL2**. This means:

- The encodings in this description are named to match the encodings of **USHLL, USHLL2**.
- The description of **USHLL, USHLL2** gives the operational pseudocode for this instruction.

\[
\begin{array}{c|cccccccccccccccc}
\hline
U & \quad 0 & Q & 1 & 0 & 1 & 1 & 1 & 1 & 0 & \neq 0000 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & Rn & Rd \\
\hline
\end{array}
\]

**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 “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\textit{SEE Advanced SIMD modified immediate}</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001\text{x}</td>
<td>4S</td>
</tr>
<tr>
<td>01\text{x}</td>
<td>2D</td>
</tr>
<tr>
<td>1\text{xxx}</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>\textit{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>001\text{x}</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001\text{x}</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01\text{x}</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01\text{x}</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1\text{xxx}</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

The description of **USHLL, USHLL2** gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to 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.

```
Vn A7 A6 A5 A4 A3 A2 A1 A0
Vm B7 B6 B5 B4 B3 B2 B1 B0
```

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 Q 0 0 1 1 1 0 size 0 Rm 0 0 1 1 0 Rn 1 0 Rd
```

```
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);
ninteger datasize = if Q == '1' then 128 else 64;
ninteger elements = datasize DIV esize;
ninteger part = UInt(op);
```

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];
```

```c
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>Vd</th>
<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>Vm</td>
<td>B7</td>
<td>B5</td>
<td>B3</td>
<td>B1</td>
<td>A7</td>
<td>A5</td>
<td>A3</td>
<td>A1</td>
</tr>
</tbody>
</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.

```
0 0 0 0 1 1 1 0 0 1 0 1 1 0
```

UZP2 `<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

```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 `FEAT_SHA3` is implemented.

### Advanced SIMD (Armv8.2)

```plaintext
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rm</th>
<th></th>
<th></th>
<th>imm6</th>
<th></th>
<th></th>
<th>Rn</th>
<th></th>
<th></th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Operation**

```plaintext
XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

- `<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”.

```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.
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 \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.

\[
\begin{array}{ccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & \text{size} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \text{Rn} & \text{Rd}
\end{array}
\]

\texttt{XTN(2) \langle Vd \rangle, \langle Tb \rangle, \langle Vn \rangle, \langle Ta \rangle}

- \texttt{integer d = UInt(Rd)};
- \texttt{integer n = UInt(Rn)};
- \texttt{if size == '11' then UNDEFINED};
- \texttt{integer esize = 8 \ll \text{UInt(size)}};
- \texttt{integer datasize = 64};
- \texttt{integer part = UInt(Q)};
- \texttt{integer elements = datasize \div esize};

\textbf{Assembler Symbols}

- \texttt{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”:
  
  \[
  \begin{array}{c|c}
  Q & 2 \\
  \hline
  0 & \text{[absent]} \\
  1 & \text{[present]} \\
  \end{array}
  \]

- \texttt{\langle Vd \rangle} is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{\langle Tb \rangle} is an arrangement specifier, encoded in “size:Q”:
  
  \[
  \begin{array}{c|c|c}
  \text{size} & \text{Q} & \langle Tb \rangle \\
  \hline
  00 & 0 & 8B \\
  00 & 1 & 16B \\
  01 & 0 & 4H \\
  01 & 1 & 8H \\
  10 & 0 & 2S \\
  10 & 1 & 4S \\
  11 & x & \text{RESERVED} \\
  \end{array}
  \]

- \texttt{\langle Vn \rangle} is the name of the SIMD&FP source register, encoded in the "Rn" field.
- \texttt{\langle Ta \rangle} is an arrangement specifier, encoded in “size”:
  
  \[
  \begin{array}{c|c}
  \text{size} & \langle Ta \rangle \\
  \hline
  00 & 8H \\
  01 & 4S \\
  10 & 2D \\
  11 & \text{RESERVED} \\
  \end{array}
  \]
Operation

```c
CheckFPAdvSIMDEnabled64();
bias(2*datasize) operand = \text{V}[n];
bias(datasize) result;
bias(2*esize) element;

for e = 0 to elements-1
    element = \text{Elem}[\text{operand}, e, 2*esize];
    \text{Elem}[\text{result}, e, \text{esize}] = \text{element}<\text{esize}-1:0>;
\text{Vpart}[\text{d}, \text{part}] = \text{result};
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ZIP1

Zip vectors (primary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP
registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination
SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with
subsequent pairs taken alternately from each source register.

This instruction can be used with ZIP2 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

```
<table>
<thead>
<tr>
<th>Vm</th>
<th>A7</th>
<th>A6</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vn</td>
<td>B7</td>
<td>B6</td>
<td>B5</td>
<td>B4</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
</tr>
</tbody>
</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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | size| Rm | 0  | 0  | 1  | 1  | 1  | 0  | Rd |
```

ZIP1 <Vd>.<T>, <Vm>.<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);
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 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    \(\text{Elem}\{\text{result}, 2*p+0, esize\} = \text{Elem}\{\text{operand1}, base+p, esize\};\)
    \(\text{Elem}\{\text{result}, 2*p+1, esize\} = \text{Elem}\{\text{operand2}, base+p, esize\};\)
\(V[d] = \text{result};\)

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.
ZIP2

Zip vectors (secondary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with ZIP1 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

\[ \begin{array}{cccccccc}
A_7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 \\
B_7 & B_6 & B_5 & B_4 & B_3 & B_2 & B_1 & B_0 \\
\end{array} \]

\[ \text{ZIP1.8, doubleword} \quad \text{ZIP2.8, doubleword} \]

Depending on the settings in the \texttt{CPACR_EL1}, \texttt{CPTR_EL2}, and \texttt{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer 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
CheckFPAdvSIMEnabled64();

bits(datasize) operand1 = \text{\texttt{V}}[n];
bits(datasize) operand2 = \text{\texttt{V}}[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    \text{\texttt{Elem}}[\text{\texttt{result}}, 2*p+0, esize] = \text{\texttt{Elem}}[\text{\texttt{operand1}}, base+p, esize];
    \text{\texttt{Elem}}[\text{\texttt{result}}, 2*p+1, esize] = \text{\texttt{Elem}}[\text{\texttt{operand2}}, base+p, esize];

\text{\texttt{V}}[d] = \text{\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.
A64 -- SVE Instructions (alphabetic order)

**ABS:** Absolute value (predicated).

**ADD (immediate):** Add immediate (unpredicated).

**ADD (vectors, predicated):** Add vectors (predicated).

**ADD (vectors, unpredicated):** Add vectors (unpredicated).

**ADDPL:** Add multiple of predicate register size to scalar register.

**ADDVL:** Add multiple of vector register size to scalar register.

**ADR:** Compute vector address.

**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).

**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.

**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).

CMP<cc> (immediate): Compare vector to immediate.
CMP<cc> (vectors): Compare vectors.
CMP<cc> (wide elements): Compare vector to 64-bit wide elements.

CMPL (vectors): Compare signed less than or equal to vector, setting the condition flags: an alias of CMP<cc> (vectors).
CMPO (vectors): Compare unsigned lower than vector, setting the condition flags: an alias of CMP<cc> (vectors).
CMPS (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).

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).

EQR: 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.

FABD: Floating-point absolute difference (predicated).
**FABS**: Floating-point absolute value (predicated).

**FAC<cc>**: Floating-point absolute compare vectors.

**FACLE**: Floating-point absolute compare less than or equal: an alias of FAC<cc>.

**FACLT**: Floating-point absolute compare less than: an alias of FAC<cc>.

**FADD (immediate)**: Floating-point add immediate (predicated).

**FADD (vectors, predicated)**: Floating-point add vector (predicated).

**FADD (vectors, unpredicated)**: Floating-point add vector (unpredicated).

**FADDA**: Floating-point add strictly-ordered reduction, accumulating in scalar.

**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).

**FCVTZS**: Floating-point convert to signed integer, rounding toward zero (predicated).

**FCVTZU**: Floating-point convert to unsigned integer, rounding toward zero (predicated).

**FDIV**: Floating-point divide by vector (predicated).

**FDIVR**: Floating-point reversed divide by vector (predicated).

**FDUP**: Broadcast 8-bit floating-point immediate to vector elements (unpredicated).

**FEXPA**: Floating-point exponential accelerator.

**FMAD**: Floating-point fused multiply-add vectors (predicated), writing multiplicand \( Z_{dn} = Z_a + Z_{dn} \times Z_m \).

**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).

**FMAXNMV**: Floating-point maximum number recursive reduction to scalar.

**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).

**FMINNNMV**: Floating-point minimum number recursive reduction to scalar.
**FMINV**: Floating-point minimum recursive reduction to scalar.

**FMLA (indexed)**: Floating-point fused multiply-add by indexed elements \((Zda = Zda + Zn \times Zm[\text{indexed}])\).

**FMLA (vectors)**: Floating-point fused multiply-add vectors (predicated), writing addend \([Zda = Zda + Zn \times Zm]\).

**FMLS (indexed)**: Floating-point fused multiply-subtract by indexed elements \((Zda = Zda + -Zn \times Zm[\text{indexed}])\).

**FMLS (vectors)**: Floating-point fused multiply-subtract vectors (predicated), writing addend \([Zda = Zda + -Zn \times Zm]\).

**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 \([Zdn = Za + -Zdn \times Zm]\).

**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 \([Zdn = -Za + -Zdn \times Zm]\).

**FNMLA**: Floating-point negated fused multiply-add vectors (predicated), writing addend \([Zda = -Zda + -Zn \times Zm]\).

**FNMLS**: Floating-point negated fused multiply-subtract vectors (predicated), writing addend \([Zda = -Zda + Zn \times Zm]\).

**FNMSB**: Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand \([Zdn = -Za + Zdn \times Zm]\).

**FRECPE**: Floating-point reciprocal estimate (unpredicated).

**FRECPS**: Floating-point reciprocal step (unpredicated).

**FRECPX**: Floating-point reciprocal exponent (predicated).

**FRINT<r>**: Floating-point round to integral value (predicated).

**FRSROTE**: Floating-point reciprocal square root estimate (unpredicated).

**FRSROTS**: Floating-point reciprocal square root step (unpredicated).

**FScale**: Floating-point adjust exponent by vector (predicated).

**FSORT**: Floating-point square root (predicated).

**FSUB (immediate)**: Floating-point subtract immediate (predicated).

**FSUB (vectors, predicated)**: Floating-point subtract vectors (predicated).

**FSUB (vectors, unpredicated)**: Floating-point subtract vectors (unpredicated).

**FSUBR (immediate)**: Floating-point reversed subtract from immediate (predicated).

**FSUBR (vectors)**: Floating-point reversed subtract vectors (predicated).

**FTMAD**: Floating-point trigonometric multiply-add coefficient.
**A64 -- SVE Instructions (alphabetic order)**

**FTSMUL**: Floating-point trigonometric starting value.

**FTSSEL**: Floating-point trigonometric select coefficient.

**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 (scalar 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 sixteen bytes (immediate index).

**LD1ROB (scalar plus scalar)**: Contiguous load and replicate sixteen bytes (scalar index).

**LD1ROD (scalar plus immediate)**: Contiguous load and replicate two doublewords (immediate index).

**LD1ROD (scalar plus scalar)**: Contiguous load and replicate two doublewords (scalar index).

**LD1ROH (scalar plus immediate)**: Contiguous load and replicate eight halfwords (immediate index).
**LD1RH (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).

**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).

**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).

**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).

**LD2B (scalar plus immediate)**: Contiguous load two-byte structures to two vectors (immediate index).

**LD2B (scalar plus scalar)**: Contiguous load two-byte structures to two vectors (scalar index).

**LD2D (scalar plus immediate)**: Contiguous load two-doubleword structures to two vectors (immediate index).

**LD2D (scalar plus scalar)**: Contiguous load two-doubleword structures to two vectors (scalar index).

**LD2H (scalar plus immediate)**: Contiguous load two-halfword structures to two vectors (immediate index).

**LD2H (scalar plus scalar)**: Contiguous load two-halfword structures to two vectors (scalar index).

**LD2W (scalar plus immediate)**: Contiguous load two-word structures to two vectors (immediate index).

**LD2W (scalar plus scalar)**: Contiguous load two-word structures to two vectors (scalar index).

**LD3B (scalar plus immediate)**: Contiguous load three-byte structures to three vectors (immediate index).

**LD3B (scalar plus scalar)**: Contiguous load three-byte structures to three vectors (scalar index).

**LD3D (scalar plus immediate)**: Contiguous load three-doubleword structures to three vectors (immediate index).

**LD3D (scalar plus scalar)**: Contiguous load three-doubleword structures to three vectors (scalar index).

**LD3H (scalar plus immediate)**: Contiguous load three-halfword structures to three vectors (immediate index).

**LD3H (scalar plus scalar)**: Contiguous load three-halfword structures to three vectors (scalar index).
LD3H (scalar plus scalar): Contiguous load three-halfword structures to three vectors (scalar index).
LD3W (scalar plus immediate): Contiguous load three-word structures to three vectors (immediate index).
LD3W (scalar plus scalar): Contiguous load three-word structures to three vectors (scalar index).
LD4B (scalar plus immediate): Contiguous load four-byte structures to four vectors (immediate index).
LD4B (scalar plus scalar): Contiguous load four-byte structures to four vectors (scalar index).
LD4D (scalar plus immediate): Contiguous load four-doubleword structures to four vectors (immediate index).
LD4D (scalar plus scalar): Contiguous load four-doubleword structures to four vectors (scalar index).
LD4H (scalar plus immediate): Contiguous load four-halfword structures to four vectors (immediate index).
LD4H (scalar plus scalar): Contiguous load four-halfword structures to four vectors (scalar index).
LD4W (scalar plus immediate): Contiguous load four-word structures to four vectors (immediate index).
LD4W (scalar plus scalar): Contiguous load four-word structures to four vectors (scalar index).
LDFF1B (scalar plus scalar): Contiguous load first-fault unsigned bytes to vector (scalar index).
LDFF1B (scalar plus vector): Gather load first-fault unsigned bytes to vector (vector index).
LDFF1B (vector plus immediate): Gather load first-fault unsigned bytes to vector (immediate index).
LDFF1D (scalar plus scalar): Contiguous load first-fault doublewords to vector (scalar index).
LDFF1D (scalar plus vector): Gather load first-fault doublewords to vector (vector index).
LDFF1D (vector plus immediate): Gather load first-fault doublewords to vector (immediate index).
LDFF1H (scalar plus scalar): Contiguous load first-fault unsigned halfwords to vector (scalar index).
LDFF1H (scalar plus vector): Gather load first-fault unsigned halfwords to vector (vector index).
LDFF1H (vector plus immediate): Gather load first-fault unsigned halfwords to vector (immediate index).
LDFF1SB (scalar plus scalar): Contiguous load first-fault signed bytes to vector (scalar index).
LDFF1SB (scalar plus vector): Gather load first-fault signed bytes to vector (vector index).
LDFF1SB (vector plus immediate): Gather load first-fault signed bytes to vector (immediate index).
LDFF1SH (scalar plus scalar): Contiguous load first-fault signed halfwords to vector (scalar index).
LDFF1SH (scalar plus vector): Gather load first-fault signed halfwords to vector (vector index).
LDFF1SH (vector plus immediate): Gather load first-fault signed halfwords to vector (immediate index).
LDFF1SW (scalar plus scalar): Contiguous load first-fault signed words to vector (scalar index).
LDFF1SW (scalar plus vector): Gather load first-fault signed words to vector (vector index).
LDFF1SW (vector plus immediate): Gather load first-fault signed words to vector (immediate index).
LDFF1W (scalar plus scalar): Contiguous load first-fault unsigned words to vector (scalar index).
LDFF1W (scalar plus vector): Gather load first-fault unsigned words to vector (vector index).
LDFF1W (vector plus immediate): Gather load first-fault unsigned words to vector (immediate index).
LDNF1B: Contiguous load non-fault unsigned bytes to vector (immediate index).
LDNF1D: Contiguous load non-fault doublewords to vector (immediate index).
LDNF1H: Contiguous load non-fault unsigned halfwords to vector (immediate index).
LDNF1SB: Contiguous load non-fault signed bytes to vector (immediate index).
**LDNF1SH**: Contiguous load non-fault signed halfwords to vector (immediate index).

**LDNF1SW**: Contiguous load non-fault signed words to vector (immediate index).

**LDNF1W**: Contiguous load non-fault unsigned words to vector (immediate index).

**LDNT1B (scalar plus immediate)**: Contiguous load non-temporal bytes to vector (immediate index).

**LDNT1B (scalar plus scalar)**: Contiguous load non-temporal bytes to vector (scalar index).

**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).

**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).

**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).

**LDR (predicate)**: Load predicate register.

**LDR (vector)**: Load vector register.

**LSL (immediate, predicated)**: Logical shift left by immediate (predicated).

**LSL (immediate, unpredicated)**: Logical shift left by immediate (unpredicated).

**LSL (vectors)**: Logical shift left by vector (predicated).

**LSL (wide elements, predicated)**: Logical shift left by 64-bit wide elements (predicated).

**LSL (wide elements, unpredicated)**: Logical shift left by 64-bit wide elements (unpredicated).

**LSLR**: Reversed logical shift left by vector (predicated).

**LSR (immediate, predicated)**: Logical shift right by immediate (predicated).

**LSR (immediate, unpredicated)**: Logical shift right by immediate (unpredicated).

**LSR (vectors)**: Logical shift right by vector (predicated).

**LSR (wide elements, predicated)**: Logical shift right by 64-bit wide elements (predicated).

**LSR (wide elements, unpredicated)**: Logical shift right by 64-bit wide elements (unpredicated).

**LSRR**: Reversed logical shift right by vector (predicated).

**MAD**: Multiply-add vectors (predicated), writing multiplicand \([Zdn = Za + Zdn \times Zm]\).

**MLA**: Multiply-add vectors (predicated), writing addend \([Zda = Zda + Zn \times Zm]\).

**MLS**: Multiply-subtract vectors (predicated), writing addend \([Zda = Zda - Zn \times Zm]\).

**MOV (bitmask immediate)**: Move logical bitmask immediate to vector (unpredicated): an alias of DUPM.

**MOV (immediate, predicated, merging)**: Move signed integer immediate to vector elements (merging): an alias of CPY (immediate, merging).

**MOV (immediate, predicated, zeroing)**: Move signed integer immediate to vector elements (zeroing): an alias of CPY (immediate, zeroing).

**MOV (immediate, unpredicated)**: Move signed immediate to vector elements (unpredicated): an alias of DUP (immediate).

**MOV (predicate, predicated, merging)**: Move predicates (merging): an alias of SEL (predicates).

**MOV (predicate, predicated, zeroing)**: Move predicates (zeroing): an alias of AND, ANDS (predicates).

**MOV (predicate, unpredicated)**: Move predicate (unpredicated): an alias of ORR, ORRS (predicates).
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, unpredicated): Move indexed element or SIMD&FP scalar to vector (unpredicated): an alias of DUP (indexed).


MOVPREFIX (predicated): Move prefix (predicated).

MOVPREFIX (unpredicated): Move prefix (unpredicated).

MOVPS (predicated): Move predicates (zeroing), setting the condition flags: an alias of AND, ANDS (predicates).

MOVPS (unpredicated): 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 (vectors): Multiply vectors (predicated).

NAND, NANDS: Bitwise NAND predicates.

NEG: Negate (predicated).

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).

OR (immediate): Bitwise inclusive OR with inverted immediate (unpredicated): an alias of ORR (immediate).

OR, ORS (predicates): Bitwise inclusive OR inverted predicate.

OR (immediate): Bitwise inclusive OR with immediate (unpredicated).

OR (vectors, predicated): Bitwise inclusive OR vectors (predicated).

OR (vectors, unpredicated): Bitwise inclusive OR vectors (unpredicated).

OR, ORS (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.

PNEXT: Find next active predicate.

PRFS (scalar plus immediate): Contiguous prefetch bytes (immediate index).

PRFS (scalar plus scalar): Contiguous prefetch bytes (scalar index).

PRFS (scalar plus vector): Gather prefetch bytes (scalar plus vector).

PRFS (vector plus immediate): Gather prefetch bytes (vector plus immediate).

PRFS (scalar plus immediate): Contiguous prefetch doublewords (immediate index).

PRFS (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 (vector plus immediate): Contiguous prefetch halfwords (vector plus immediate).
PRFH (scalar plus immediate): Contiguous prefetch halfwords (immediate 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.
RBIT: Reverse bits (predicated).
RDFFR (unpredicated): Read the first-fault register.
RDFFR, RDFFRS (predicated): Return predicate of successfully 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).
SABD: Signed absolute difference (predicated).
SADDV: Signed add reduction to scalar.
SCVTF: Signed integer convert to floating-point (predicated).
SDIV: Signed divide (predicated).
SDIVR: Signed reversed divide (predicated).
SDOT (indexed): Signed integer indexed dot product.
SDOT (vectors): Signed integer dot product.
SEL (predicates): Conditionally select elements from two predicates.
SEL (vectors): Conditionally select elements from two vectors.
SETFFR: Initialise the first-fault register to all true.
SMAX (immediate): Signed maximum with immediate (unpredicated).
SMAX (vectors): Signed maximum vectors (predicated).
SMAXV: Signed maximum reduction to scalar.
SMIN (immediate): Signed minimum with immediate (unpredicated).
SMIN (vectors): Signed minimum vectors (predicated).
SMINV: Signed minimum reduction to scalar.
SMULH: Signed multiply returning high half (predicated).

SPLICE: Splice two vectors under predicate control.

SQADD (immediate): Signed saturating add immediate (unpredicated).

SQADD (vectors): Signed saturating add vectors (unpredicated).

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.


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.

SQSUB (immediate): Signed saturating subtract immediate (unpredicated).

SQSUB (vectors): Signed saturating subtract vectors (unpredicated).

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 (scalar plus vector): Scatter store halfwords from a vector (vector index).

ST1H (vector plus immediate): Scatter store halfwords from a vector (immediate index).
ST1W (scalar plus immediate): Contiguous store words from vector (immediate index).
ST1W (scalar plus scalar): Contiguous store words from vector (scalar index).
ST1W (scalar plus vector): Scatter store words from a vector (vector index).
ST1W (vector plus immediate): Scatter store words from a vector (immediate index).
ST2B (scalar plus immediate): Contiguous store two-byte structures from two vectors (immediate index).
ST2B (scalar plus scalar): Contiguous store two-byte structures from two vectors (scalar index).
ST2D (scalar plus immediate): Contiguous store two-doubleword structures from two vectors (immediate index).
ST2D (scalar plus scalar): Contiguous store two-doubleword structures from two vectors (scalar index).
ST2H (scalar plus immediate): Contiguous store two-halfword structures from two vectors (immediate index).
ST2H (scalar plus scalar): Contiguous store two-halfword structures from two vectors (scalar index).
ST2W (scalar plus immediate): Contiguous store two-word structures from two vectors (immediate index).
ST2W (scalar plus scalar): Contiguous store two-word structures from two vectors (scalar index).
ST3B (scalar plus immediate): Contiguous store three-byte structures from three vectors (immediate index).
ST3B (scalar plus scalar): Contiguous store three-byte structures from three vectors (scalar index).
ST3D (scalar plus immediate): Contiguous store three-doubleword structures from three vectors (immediate index).
ST3D (scalar plus scalar): Contiguous store three-doubleword structures from three vectors (scalar index).
ST3H (scalar plus immediate): Contiguous store three-halfword structures from three vectors (immediate index).
ST3H (scalar plus scalar): Contiguous store three-halfword structures from three vectors (scalar index).
ST3W (scalar plus immediate): Contiguous store three-word structures from three vectors (immediate index).
ST3W (scalar plus scalar): Contiguous store three-word structures from three vectors (scalar index).
ST4B (scalar plus immediate): Contiguous store four-byte structures from four vectors (immediate index).
ST4B (scalar plus scalar): Contiguous store four-byte structures from four vectors (scalar index).
ST4D (scalar plus immediate): Contiguous store four-doubleword structures from four vectors (immediate index).
ST4D (scalar plus scalar): Contiguous store four-doubleword structures from four vectors (scalar index).
ST4H (scalar plus immediate): Contiguous store four-halfword structures from four vectors (immediate index).
ST4H (scalar plus scalar): Contiguous store four-halfword structures from four vectors (scalar index).
ST4W (scalar plus immediate): Contiguous store four-word structures from four vectors (immediate index).
ST4W (scalar plus scalar): Contiguous store four-word structures from four vectors (scalar index).
STNT1B (scalar plus immediate): Contiguous store non-temporal bytes from vector (immediate index).
STNT1B (scalar plus scalar): Contiguous store non-temporal bytes from vector (scalar index).
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).
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).
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).
**A64 -- SVE Instructions (alphabetic order)**

- **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).
- **SUBR (immediate):** Reversed subtract from immediate (unpredicated).
- **SUBR (vectors):** Reversed subtract vectors (predicated).
- **SUNPKHI, SUNPKLO:** Signed unpack and extend half of vector.
- **SXTB, SXTH, SXTW:** Signed byte / halfword / word extend (predicated).
- **TBL:** Programmable table lookup in single vector table.
- **TRN1, TRN2 (predicates):** Interleave even or odd elements from two predicates.
- **TRN1, TRN2 (vectors):** Interleave even or odd elements from two vectors.
- **UABD:** Unsigned absolute difference (predicated).
- **UADDV:** Unsigned add reduction to scalar.
- **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.
- **UMAX (immediate):** Unsigned maximum with immediate (unpredicated).
- **UMAX (vectors):** Unsigned maximum vectors (predicated).
- **UMAXV:** Unsigned maximum reduction to scalar.
- **UMIN (immediate):** Unsigned minimum with immediate (unpredicated).
- **UMIN (vectors):** Unsigned minimum vectors (predicated).
- **UMINV:** Unsigned minimum reduction to scalar.
- **UMULH:** Unsigned multiply returning high half (predicated).
- **UQADD (immediate):** Unsigned saturating add immediate (unpredicated).
- **UQADD (vectors):** 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.
UQSUB (immediate): Unsigned saturating subtract immediate (unpredicated).
UQSUB (vectors): Unsigned saturating subtract vectors (unpredicated).
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.
WHILELE: While incrementing signed scalar less than or equal to scalar.
WHILELO: While incrementing unsigned scalar lower than scalar.
WHILELS: While incrementing unsigned scalar lower or same as scalar.
WHILELT: While incrementing signed scalar less than scalar.
WRFFR: Write the first-fault register.
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.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    integer element = SInt(Elem[operand, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        element = Abs(element);
        Elem[result, e, esize] = element<esize-1:0>;
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
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":
  - 00: B
  - 01: H
  - 10: S
  - 11: D
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 + element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ADD (immediate)

Add immediate (unpredicated).

Add an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

```
>             <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(uimm8);
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
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 + imm;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

```
<table>
<thead>
<tr>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
ADD <Zd>..<T>, <Zn>..<T>, <Zm>..<T>
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 + element2;
Z[d] = result;
```
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 |

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;

Internal version only: isa_v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADDVL

Add multiple of vector register size to scalar register.

Add the current vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | Rn | 0 | 1 | 0 | 1 | 0 | imm6 | Rd |

ADDVL <Xd|SP>, <Xn|SP>, #<imm>

if !HaveSVE() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler Symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) result = operand1 + (imm * (VL DIV 8));
if d == 31 then
    SP[] = result;
else
    X[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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
```

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 signed 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 0 1 Zm 1 0 1 0 msz Zn Zd
```

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

```
31 30 29 28 27 26 25 24 23 22 21 20 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 Zm 1 0 1 0 msz Zn Zd
```

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":

ADR
<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.

\(<\text{mod}>\)  Is the index extend and shift specifier, encoded in "msz":

<table>
<thead>
<tr>
<th>msz</th>
<th>&lt;\text{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>

\(<\text{amount}>\)  Is the index shift amount, encoded in "msz":

<table>
<thead>
<tr>
<th>msz</th>
<th>&lt;\text{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**

```plaintext
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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AND, ANDS (predicates)

Bitwise AND predicates.

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the aliases MOV (predicated), and MOV (predicate, predicated, zeroing).

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 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 = 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 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 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 (predicated)</td>
<td>$S == '1' &amp;&amp; Pn == Pm</td>
</tr>
<tr>
<td>MOV (predicate, predicated, zeroing)</td>
<td>$S == '0' &amp;&amp; Pn == Pm</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \div \texttt{esize};
bits(PL) mask = \texttt{P}[g];
bits(PL) operand1 = \texttt{P}[n];
bits(PL) operand2 = \texttt{P}[m];
bits(PL) result;

for e = 0 to elements-1
  bit element1 = \texttt{ElemP}[operand1, e, esize];
  bit element2 = \texttt{ElemP}[operand2, e, esize];
  if \texttt{ElemP}[mask, e, esize] == '1' then
    \texttt{ElemP}[result, e, esize] = element1 AND element2;
  else
    \texttt{ElemP}[result, e, esize] = '0';

if setflags then
  PSTATE.<N,Z,C,V> = \texttt{PredTest}(mask, result, esize);
P[d] = result;
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 0 1 0 0 size 0 1 1 0 1 0 0 0 0 Pg Zm Zdn

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
AND (immediate)

Bitwise AND with immediate (unpredicated).

Bitwise AND an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction BIC (immediate).

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

AND <Zdn>.<T>, <Zdn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
i = UInt(Zdn);
b = (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>”:

<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();
i = VL DIV 64;
b = Z[dn];
for e = 0 to elements-1
    b = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 AND imm;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

```
0 0 0 0 0 1 0 0 0 0 1 Zm 0 0 1 1 0 0 Zn Zd
```

AND <Zd>.D, <Zn>.D, <Zm>.D

if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();

bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];

Z[d] = operand1 AND operand2;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 1 0 0 | size | 0 1 1 | 0 1 | 0 0 0 1 | Pg | Zn | Vd |

\( \text{ANDV} \langle V \rangle, \langle d \rangle, \langle Pg \rangle, \langle Zn \rangle, \langle T \rangle \)

if !\( \text{HaveSVE}() \) then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
in integer n = UInt(Zn);
in integer d = UInt(Vd);

**Assembler Symbols**

\( \langle V \rangle \) is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>( \langle V \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>

\( \langle d \rangle \) is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.
\( \langle Pg \rangle \) is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\( \langle Zn \rangle \) is the name of the source scalable vector register, encoded in the "Zn" 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>

**Operation**

\( \text{CheckSVEEnabled}() \);
in integer elements = \( \text{VL} \) DIV esize;
bits(PL) mask = \( P[g] \);
bits(VL) operand = \( Z[n] \);
bits(esize) result = \( \text{Ones} \)(esize);

for e = 0 to elements-1
  if \( \text{Elem}[\text{mask}, e, \text{esize}] \) == ‘1’ then
    result = result AND \( \text{Elem}[\text{operand}, e, \text{esize}] \);
\( V[d] = \text{result} \);
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.

```
0 0 0 0 0 0 1 0 0 | tszh 0 0 0 0 0 0 0 1 0 0 | Pg  | tSZl | imm3 | Zdn
```

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);
```

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 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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

```
31 30 29 28 27 26 25 24 23 22 21 20 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 0 1 0 0 | Pg | Zm | Zdn

ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D
```

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in "size":

```
size <T>
00 B
01 H
10 S
11 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**

```assembly
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**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ASR (vectors)

Arithmetic shift right by vector (predicated).

Shift right, preserving the sign bit, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

```
| 0  0  0  0  1  0  0 | size     | 0  | 1  | 0  | 0  | 0  | 1  | 0  |
|----------------------|----------|----|----|----|----|----|----|
| Pg                   | Zm       | Zdn|
```

ASR <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>,<T>

```
if !HaveSVE() then UNDEFINED;
integer size = 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 size;
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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ASR (immediate, unpredicated)

Arithmetic shift right by immediate (unpredicated).

Shift right by immediate, preserving the sign bit, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Assembling 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 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;
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.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>
```

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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>
```

ASRD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tshz:tshl;
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(tsh;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 “tshz:tshl”:

<table>
<thead>
<tr>
<th>tshz</th>
<th>tshl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsh:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  if ElemP[mask, e, esize] == '1' then
    if element1 < 0 then
      element1 = element1 + ((1 << shift) - 1);
    Elem[result, e, esize] = (element1 >> shift)<<(esize-1:0>);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

ASRD
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

<table>
<thead>
<tr>
<th>0 0 0 0</th>
<th>0 1 0 1</th>
<th>0 1 0</th>
<th>0 1 0 0</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
</table>

ASRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = ASR(element2, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**BIC (Immediate)**

Bitwise clear bits using immediate (unpredicated).

Bitwise clear bits using immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of `AND (Immediate)`. This means:

- The encodings in this description are named to match the encodings of `AND (Immediate)`.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of `AND (Immediate)` gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>imm13</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1 1 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BIC <Zdn>.<T>, <Zdn>.<T>, #<const>**

is equivalent to

**AND <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)**

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<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**

The description of `AND (Immediate)` gives the operational pseudocode for this instruction.

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**BIC, BICS (predicates)**

Bitwise clear predicates.

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

**Not setting the condition flags**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 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 = FALSE;

**Setting the condition flags**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |


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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
    P[d] = result;
```

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 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0</td>
<td>0 1 1</td>
<td>0 1 1</td>
<td>0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

BIC <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 (NOT element2);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

| 31 30 29 28 27 26 25 24 23 22 21 20 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              | Zm                               | Zn                               |
| 0 0 1 1 0 0                  | Zd                               |                                  |

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);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BRKA, BRKAS

Break after first true condition.

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|--------|--------|--------|
|                                        | B      | S      |

BRKA <Pd>.B, <Pg>/<ZM>, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;

Setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|--------|--------|--------|
|                                        | B      | S      |

BRKAS <Pd>.B, <Pg>/Z, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<ZM> Is the 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
    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;
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

\[
\begin{array}{cccccccccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & | & Pg & 0 & | & Pn & M & | & Pd & \\
\end{array}
\]

BRKB <Pd>.B, <Pg>/<ZM>, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;

### Setting the condition flags

\[
\begin{array}{cccccccccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & | & Pg & 0 & | & Pn & 0 & | & Pd & M \\
\end{array}
\]

BRKBS <Pd>.B, <Pg>/Z, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;

### Assembler Symbols

- **<Pd>**  
  Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Pg>**  
  Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- **<ZM>**  
  Is the predication qualifier, encoded in “M”:
  \[
  \begin{array}{c|c}
  M & <ZM> \\
  0 & Z \\
  1 & M \\
  \end{array}
  \]
- **<Pn>**  
  Is the name of the source scalable predicate register, encoded in the "Pn" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) 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;
        ElemP[result, e, esize] = if !break then '1' else '0';
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';
    end if

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
end if
P[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 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 0</th>
<th>Pn 0</th>
<th>Pdm</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
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>
<th>Pg 0</th>
<th>Pn 0</th>
<th>Pdm</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = TRUE;

Assembler Symbols

\(<Pdm>\) Is the name of the second source and destination scalable predicate register, encoded in the “Pdm” field.

\(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.

\(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[dm];
bits(PL) result;
if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros();
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm] = result;
BRKPA, BRKPAS

Break after first true condition, propagating from previous partition.

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 0 0 0 1 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</td>
</tr>
</tbody>
</table>

S B


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 1 0 0 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 0 0</td>
</tr>
</tbody>
</table>

S B


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
      ElemP[result, e, 8] = if last then '1' else '0';
      last = last && (ElemP[operand2, e, 8] == '0');
  else
      ElemP[result, e, 8] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
BRKPB, BRKPBS

Break before first true condition, propagating from previous partition.

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

<table>
<thead>
<tr>
<th>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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

<table>
<thead>
<tr>
<th>31</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>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>

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;
```

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.

\[
\begin{array}{cccccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & \text{Pg} & \text{Zm} & \text{Rdn} & B
\end{array}
\]

CLASTA \(\langle R\rangle<dn>, \langle Pg\rangle, \langle R\rangle<dn>, \langle Zm\rangle.<T>\)

if !\text{HaveSVE}() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
integer csize = if esize < 64 then 32 else 64;
boolean isBefore = FALSE;

Assembler Symbols

\(<R>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;R&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

\(<dn>\) Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the source scalable vector register, encoded in the "Zm" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

\text{CheckSVEEnabled}();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = X[dn];
bits(VL) operand2 = Z[m];
bits(csize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = ZeroExtend(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
        result = ZeroExtend(Elem(operand2, last, esize));

X[dn] = result;
CLASTA (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.

```
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 1 0 1 0 1 0 1 0 1 0 0 8 7 6 5 4 3 2 1 0
Pg Zm Vdn
```

CLASTA \(<V><dn>, <Pg>, <V><dn>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 \(<\text{UInt}(size));
integer g = \text{UInt}(Pg);
integer dn = \text{UInt}(Vdn);
integer m = \text{UInt}(Zm);
boolean isBefore = FALSE;

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>

<table>
<thead>
<tr>
<th>&lt;dn&gt;</th>
<th>Is the number [0-31] of the source and destination SIMD&amp;FP register, encoded in the &quot;Vdn&quot; field.</th>
</tr>
</thead>
<tbody>
<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;Zm&gt;</td>
<td>Is the name of the source scalable vector register, encoded in the &quot;Zm&quot; field.</td>
</tr>
<tr>
<td>&lt;T&gt;</td>
<td>Is the size specifier, encoded in “size”:</td>
</tr>
<tr>
<td>size</td>
<td>&lt;T&gt;</td>
</tr>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

```
CheckSVEEnabled();
integer elements = \text{VL} \text{DIV} esize;
\text{bits}(PL) mask = P[g];
\text{bits}(esize) operand1 = V[dn];
\text{bits}(VL) operand2 = Z[m];
\text{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 = \text{Elem}(operand2, last, esize);
V[dn] = result;
```
CLASTA (vectors)

Conditionally extract element after last to vector register.

From the second source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  |
| Pg | Zm | Zdn |

CLASTA \(<Zdn>.<T>, \langle Pg\rangle, <Zdn>.<T>, <Zm>.<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer g = \text{UInt}(Pg);
integer dn = \text{UInt}(Zdn);
integer m = \text{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

\(\text{CheckSVEEnabled}();\)
integer elements = \text{VL} \text{DIV} esize;
\text{bits(PL)} mask = \text{P}[g];
\text{bits(VL)} operand1 = \text{Z}[dn];
\text{bits(VL)} operand2 = \text{Z}[m];
\text{bits(VL)} result;
integer last = \text{LastActiveElement}(mask, esize);
if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
    if last >= elements then last = 0;
    for e = 0 to elements-1
        \text{Elem}[result, e, esize] = \text{Elem}[operand2, last, esize];
\text{Z}[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CLASTB (scalar)

Conditionally extract last element to general-purpose register.

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.

![Illustration of register values]

CLASTB 

```python
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
integer csize = if esize < 64 then 32 else 64;
boolean isBefore = TRUE;
```

Assembler Symbols

- `<R>` Is a width specifier, encoded in “size”:
  ```
  size   <R>
  01   W
  x0   W
  11   X
  ```

- `<dn>` Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Zm>` Is the name of the source scalable vector register, encoded in the "Zm" field.

- `<T>` Is the size specifier, encoded in “size”:
  ```
  size   <T>
  00   B
  01   H
  10   S
  11   D
  ```

Operation

```python
CheckSVEEnabled();
inelements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = X[dn];
bits(VL) operand2 = Z[m];
bits(csize) result;
ilast = 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));
in[dn] = result;
```
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 1 0 1 0 1 1 1 0 0 | Pg | Zm | Vdn |
```

CLASTB \(<V><dn>, <Pg>, <V><dn>, <Zm>.\<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 \(<\text{UInt}(size)\);
integer g = \text{UInt}(Pg);
integer dn = \text{UInt}(Vdn);
integer m = \text{UInt}(Zm);
boolean isBefore = TRUE;

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size (&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 B</td>
</tr>
<tr>
<td>01 H</td>
</tr>
<tr>
<td>10 S</td>
</tr>
<tr>
<td>11 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 (&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 B</td>
</tr>
<tr>
<td>01 H</td>
</tr>
<tr>
<td>10 S</td>
</tr>
<tr>
<td>11 D</td>
</tr>
</tbody>
</table>
```

Operation

\(\text{CheckSVEEnabled}();\)
integer elements = \(\text{VL DIV esize};\)
bits(PL) mask = \(P[g];\)
bits(esize) operand1 = \(V[dn];\)
bits(VL) operand2 = \(Z[m];\)
bits(esize) result;
integer last = \(\text{LastActiveElement}(mask, esize);\)
if last < 0 then
  result = \(\text{ZeroExtend}(operand1);\)
else
  if \(!\text{isBefore then}\)
    last = last + 1;
  if last \(\geq\) elements then last = 0;
  result = \(\text{Elem}(operand2, last, esize];\)
\(V[dn] = result;\)
CLASTB (vectors)

Conditionally extract last element to vector register.

From the second source vector register extract the last active element, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  |

CLASTB <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 = 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;
integer last = LastActiveElement(mask, esize);
if last < 0 then
  result = operand1;
else
  if !isBefore then
    last = last + 1;
  if last >= elements then last = 0;
  for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand2, last, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

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":
  - 00: B
  - 01: H
  - 10: S
  - 11: D
- <Pg> is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();

integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

for e = 0 to elements-1

bits(esize) element = Elem[operand, e, esize];
if ElemP[mask, e, esize] == '1'

bits(esize) result = CountLeadingSignBits(element)<esize-1:0>);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction is undefined if:

• The MOVPRFX instruction does not refer to the same destination register as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The MOVPRFX instruction must be predicated using the same governing predicate register.
• The MOVPRFX instruction specifies the same destination register as this instruction.
• The destination register cannot refer to architectural register state referenced by any other source operand.

The assembler instruction must not refer to any architectural registers that are referenced by any other source operand.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CLZ

Count leading zero bits (predicated).

Count leading zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |
| size | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

CLZ <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = CountLeadingZeroBits(element)<esize-1:0>;
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>imm5</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>Pg</th>
<th>Zn</th>
<th>0</th>
<th>Pd</th>
</tr>
</thead>
</table>

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

**Greater than**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>imm5</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Pg</th>
<th>Zn</th>
<th>1</th>
<th>Pd</th>
</tr>
</thead>
</table>

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

**Greater than or equal**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>imm5</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Pg</th>
<th>Zn</th>
<th>0</th>
<th>Pd</th>
</tr>
</thead>
</table>

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
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;

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;

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;

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;

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;
CMPLE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Lower

\[
\begin{array}{cccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 1 & \text{imm7} & 1 & \text{Pg} & \text{Zn} & 0 & \text{Pd} \\
\end{array}
\]

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 = UInt(imm7);
boolean unsigned = TRUE;

Lower or same

\[
\begin{array}{ccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 1 & \text{imm7} & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

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

\[
\begin{array}{ccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & \text{imm5} & 1 & 0 & 0 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Is the name of the source scalable vector register, encoded in the "Zn" field.

For the equal, greater than, greater than or equal, less than, less than or equal and not equal variant: is the signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

For the higher, higher or same, lower and lower or same variant: is the unsigned immediate operand, in the range 0 to 127, encoded in the “imm7” field.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(PL) result;
for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   if ElemP[mask, e, esize] == '1' then
      boolean cond;
      case op of
         when Cmp_EQ cond = element1 == imm;
         when Cmp_NE cond = element1 != imm;
         when Cmp_GE cond = element1 >= imm;
         when Cmp_LT cond = element1 <  imm;
         when Cmp_GT cond = element1 >  imm;
         when Cmp_LE cond = element1 <= imm;
           ElemP[result, e, esize] = if cond then '1' else '0';
      else
         ElemP[result, e, esize] = '0';
   end
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMP<cc> (wide elements)

Compare vector to 64-bit wide elements.

Compare active integer elements in the first source vector with overlapping 64-bit doubleword elements in the second source vector and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, HI, HS, LE, LO, LS, LT or NE.

It has encodings from 10 classes: **Equal**, **Greater than**, **Greater than or equal**, **Higher**, **Higher or same**, **Less than**, **Less than or equal**, **Lower**, **Lower or same** and **Not equal**

### Equal

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  |

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```plaintext
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  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  |

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```plaintext
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  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  |
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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 0 0 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 0 0 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = FALSE;
Less than or equal

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & Zm & 0 & 1 & 1 & Pg & Zn & 1 & Pd \\
\end{array}
\]

\text{CMPLE} <\text{Pd}>, <\text{T}>, <\text{Pg}>/Z, <\text{Zn}>, <\text{T}>, <\text{Zm}>

\begin{verbatim}
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = FALSE;
\end{verbatim}

Lower

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & Zm & 1 & 1 & 1 & Pg & Zn & 0 & Pd \\
\end{array}
\]

\text{CMPLT} <\text{Pd}>, <\text{T}>, <\text{Pg}>/Z, <\text{Zn}>, <\text{T}>, <\text{Zm}>

\begin{verbatim}
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = TRUE;
\end{verbatim}

Lower or same

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & Zm & 1 & 1 & 1 & Pg & Zn & 1 & Pd \\
\end{array}
\]

\text{CMPLS} <\text{Pd}>, <\text{T}>, <\text{Pg}>/Z, <\text{Zn}>, <\text{T}>, <\text{Zm}>

\begin{verbatim}
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;
\end{verbatim}

Not equal

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & Zm & 0 & 0 & 1 & Pg & Zn & 1 & Pd \\
\end{array}
\]

\text{C}MP<cc> (wide elements)

Page 1517
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer n = Uint(Zn);
integer m = Uint(Zm);
integer d = Uint(Pd);
SVECmp op = Cmp_NE;
boolean unsigned = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(PL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, (e * esize) DIV 64, 64], unsigned);
    if ElemP[mask, e, esize] == '1' then
        boolean cond;
        case op of
            when Cmp_EQ cond = element1 == element2;
            when Cmp_NE cond = element1 != element2;
            when Cmp_GE cond = element1 >= element2;
            when Cmp_LT cond = element1 <  element2;
            when Cmp_GT cond = element1 >  element2;
            when Cmp_LE cond = element1 <= element2;
                ElemP[result, e, esize] = if cond then '1' else '0';
        else
            ElemP[result, e, esize] = '0';
        end case;
    end if;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
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

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | Pd |

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

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | Pg | Zn | 1 | Pd |

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

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | Pg | Zn | 0 | Pd |
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

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

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

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_NE;
boolean unsigned = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```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], 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 if;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMPLE (vectors)

Compare signed less than or equal to vector, setting the condition flags.

Compare active signed integer elements in the first source vector being less than or equal to corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

```
  31  30  29  28  27  26  25  24  23  22  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  0  Pg  Zn  0  Pd
```

CMPLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:

```
  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.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMPLO (vectors)

Compare unsigned lower than vector, setting the condition flags.

Compare active unsigned integer elements in the first source vector being lower than corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of **CMP<cc> (vectors)**. This means:

- The encodings in this description are named to match the encodings of **CMP<cc> (vectors)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **CMP<cc> (vectors)** gives the operational pseudocode for this instruction.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

**CMPLO <Pd>..<T>, <Pg>/Z, <Zm>..<T>, <Zn>..<T>**

is equivalent to

**CMPHP <Pd>..<T>, <Pg>/Z, <Zn>..<T>, <Zm>..<T>**

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of **CMP<cc> (vectors)** gives the operational pseudocode for this instruction.
CMPLS (vectors)

Compare unsigned lower or same as vector, setting the condition flags.

Compare active unsigned integer elements in the first source vector being lower than or same as corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

```
CMPLS <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.
CMPLT (vectors)

Compare signed less than vector; setting the condition flags.

Compare active signed integer elements in the first source vector being less than corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

CMPLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.
**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 | 1 | 0 | 0 | size | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | Pg | Zn | Zd |

CNOT $<Zd>,<T>$, $<Pg>/M$, $<Zn>$. $<T>$

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

**CheckSVEEnabled();**
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
  bits(esize) element = Elem[operand, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = ZeroExtend(IsZeroBit(element), esize);
Z[d] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

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 Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = BitCount(element<esize-1:0>);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CNTB, CNTD, CNTH, CNTW

Set scalar to multiple of predicate constraint element count.

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then places the result in the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword and Word

### Byte

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 0 0 0 1 0 imm4 1 1 1 0 0 0 | pattern | Rd |
|---|---|---|

CNTB <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

### Doubleword

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 0 1 1 0 imm4 1 1 1 0 0 0 | pattern | Rd |
|---|---|---|

CNTD <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

### Halfword

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 0 1 1 0 imm4 1 1 1 0 0 0 | pattern | Rd |
|---|---|---|

CNTH <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
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  | 0  | 1  | 0  | imm4 | 1  | 1  | 1  | 0  | 0  | 0  | pattern | Rd |

size<1>size<0>

CNTW <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx0x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx0x0</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);

X[d] = (count * imm)<63:0>;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
0 0 1 0 0 1 0 1 size 1 0 0 0 0 | 0 1 0 Pg 0 | Pn | Rd
```

```
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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

\[
\begin{array}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & \text{Pg} & \text{Zn} & \text{Zd}
\end{array}
\]

COMPACT \(<\text{Zd}.<\text{T}>, <\text{Pg}>, <\text{Zn}>.<\text{T}>\)

\[
\begin{align*}
\text{if } !\text{HaveSVE}() & \text{ then UNDEFINED;} \\
\text{if size } &= \text{ '0x' then UNDEFINED;} \\
\text{integer esize } &= 8 \ll \text{UInt}(\text{size}); \\
\text{integer g } &= \text{UInt(Pg);} \\
\text{integer n } &= \text{UInt(Zn);} \\
\text{integer d } &= \text{UInt(Zd);} \\
\end{align*}
\]

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<0>”:
  \[
  \begin{array}{cc}
  \text{size<0>} & <\text{T}> \\
  0 & S \\
  1 & D \\
  \end{array}
  \]
- \(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<\text{Zn}>\) Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

\[
\text{CheckSVEEnabled}(); \\
\text{integer elements } = \text{VL} \text{DIV esize}; \\
\text{bits(PL) mask } = \text{P}[g]; \\
\text{bits(PL) operand1 } = \text{Z}[n]; \\
\text{bits(PL) result}; \\
\text{integer x } = 0; \\
\text{for e } = 0 \text{ to elements-1} \\
\quad \text{Elem}[\text{result, e, esize} ] = \text{Zeros}(); \\
\quad \text{if ElemP[mask, e, esize] } == \text{ '1'} \text{ then} \\
\quad \quad \text{bits(esize) element } = \text{Elem[operand1, e, esize]}; \\
\quad \quad \text{Elem[result, x, esize] } = \text{element}; \\
\quad \quad x = x + 1; \\
\text{Z[d] } = \text{result};
\]

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**CPY (immediate, zeroing)**

Copy signed integer immediate to vector elements (zeroing).

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0). The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias **MOV (immediate, predicated, zeroing)**.

| 31 30 29 28 27 26 25 24 23 22 21 20 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 0 | sh | imm8 | Zd |

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<imm>` Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>
**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) dest = Z[d];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = imm<esize-1:0>;
    elsif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros();

Z[d] = result;
```
**CPY (immediate, merging)**

Copy signed integer immediate to vector elements (merging).

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the aliases **FMOV (zero, predicated)**, and **MOV (immediate, predicated, merging)**.

---

**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();
    end if
end for

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**CPY (scalar)**

Copy general-purpose register to vector elements (predicated).

Copy the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias **MOV (scalar, predicated)**.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| CPY <Zd>.<T>, <Pg>/M, <R><n|SP> |
```

If `!HaveSVE()` then UNDEFINED;

integer `esize = 8 « UInt(size)`;
integer `g = UInt(Pg)`;
integer `n = UInt(Rn)`;
integer `d = UInt(Zd)`;

### Assembler Symbols

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<R>`: Is a width specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>
```

- `<n|SP>`: Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

### Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bis(PL) mask = P[g];
bis(64) operand1;
if n == 31 then
    operand1 = SP[];
else
    operand1 = X[n];
bis(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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CPY (SIMD&FP scalar)

Copy SIMD&FP scalar register to vector elements (predicated).

Copy the SIMD & floating-point scalar source register to each active element in the destination vector.Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias MOV (SIMD&FP scalar, predicated).

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 00 00 00 01 01 | size | 1 0 0 0 0 0 1 0 0 | Pg | Vn | Zd |
```

CPY <Zd>.<T>, <Pg>/M, <V><n>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Vn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<V> Is a width specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the “Vn” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = V[n];
bits(VL) result = Z[d];
for e = 0 to elements - 1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = operand1;
Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**CTERMEQ, CTERMNE**

Compare and terminate loop.

Detect termination conditions in serialized vector loops. Tests whether the comparison between the scalar source operands holds true and if not tests the state of the !LAST condition flag (C) which indicates whether the previous flag-setting predicate instruction selected the last element of the vector partition.

The Z and C condition flags are preserved by this instruction. The N and V condition flags are set as a pair to generate one of the following conditions for a subsequent conditional instruction:

* GE (N=0 & V=0): continue loop (compare failed and last element not selected);
* LT (N=0 & V=1): terminate loop (last element selected);
* LT (N=1 & V=0): terminate loop (compare succeeded);

The scalar source operands are 32-bit or 64-bit general-purpose registers of the same size.

It has encodings from 2 classes: **Equal** and **Not equal**

### Equal

```
0 0 1 0 0 1 1 sz 1 Rm 0 0 1 0 0 0 Rn 0 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

```
0 0 1 0 0 1 1 sz 1 Rm 0 0 1 0 0 0 Rn 1 0 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

**<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

```plaintext
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);
```
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 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 0 1 1 | pattern | Rdn |

\[
\text{DECB} <Xdn\\{, \langle\text{pattern}\rangle\{, \text{MUL} \#<\text{imm}\}}\}
\]

if \(\text{HaveSVE}()\) 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 1 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 0 1 1 1 | pattern | Rdn |

\[
\text{DECD} <Xdn\\{, \langle\text{pattern}\rangle\{, \text{MUL} \#<\text{imm}\}}\}
\]

if \(\text{HaveSVE}()\) 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 0 0 1 1 1 0 0 0 1 0 0 0 0 0 1 1 1 | pattern | Rdn |

\[
\text{DECH} <Xdn\\{, \langle\text{pattern}\rangle\{, \text{MUL} \#<\text{imm}\}}\}
\]

if \(\text{HaveSVE}()\) then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
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>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(64) operand1 = X[dn];

X[dn] = operand1 - (count * imm);
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

```assembly
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

```assembly
DECH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Word

```assembly
DECW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11100</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>

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**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
DECP (scalar)

Decrement scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination.

<table>
<thead>
<tr>
<th>size</th>
<th>esize</th>
<th>Pm</th>
<th>Rdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>1 0 1 1 0 1 1 0</td>
<td>0 0 1 0 0 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>

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;
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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Pm | Zdn |
```

DECP <Zdn>.<T>, <Pm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pm>` Is the name of the source scalable predicate register, encoded in the "Pm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
    if ElemP(operand2, e, esize] == '1' then
        count = count + 1;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] - count;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
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).

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;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

DUP <Zd>.<T>, <R><n|SP>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) operand;
if n == 31 then
    operand = SP[];
else
    operand = X[n];
bits(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = operand<esize-1:0>;
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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</th>
<th>19 18 17 16 15 14 13 12 11 10</th>
<th>9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>imm2</td>
<td>tsz</td>
</tr>
</tbody>
</table>

DUP <Zd>.<T>, <Zn>.<T>[<imm>]

if !HaveSVE() then UNDEFINED;
bits(7) imm = imm2:tsz;
case tsz of
  when '00000' UNDEFINED;
  when '10000' esize = 128; index = UInt(imm<6:5>);
  when '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>x1000</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

```
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;
```
**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)**.

### Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;Zd&gt;</th>
<th>Is the name of the destination scalable vector register, encoded in the &quot;Zd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;T&gt;</td>
<td>Is the size specifier, encoded in “imm13&lt;12&gt;:imm13&lt;5:0&gt;”:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOV (bitmask immediate)</strong></td>
<td><strong>SVEMoveMaskPreferred(imm13)</strong></td>
</tr>
</tbody>
</table>

### Operation

**CheckSVEEnabled();**

bits(VL) result = **Replicate**(imm);

Z[d] = result;

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
EON

Bitwise exclusive OR with inverted immediate (unpredicated).

Bitwise exclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of EOR (immediate). This means:

• The encodings in this description are named to match the encodings of EOR (immediate).
• The assembler syntax is used only for assembly, and is not used on disassembly.
• The description of EOR (immediate) gives the operational pseudocode for this instruction.

```
0 0 0 0 1 0 1 0 1 0 0 0 0 imm13 Zdn
```

EON <Zdn>.<T>, <Zdn>.<T>, #<const>

is equivalent to

EOR <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>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

The description of EOR (immediate) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
EOR, EORS (predicates)

Bitwise exclusive OR predicates.

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the aliases NOTS, and NOT (predicate).

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

### Not setting the condition flags

```
31 30 29 28 27 26 25 24 23 22 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 | | | | Pm | 0 | 1 | Pg | 1 | | | Nn | 0 | Pd | 1
```


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

### Setting the condition flags

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | | | | Pm | 0 | 1 | Pg | 1 | | | Nn | 0 | Pd | 1
```


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

### Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTS</td>
<td>Pm == Pg</td>
</tr>
<tr>
<td>NOT (predicate)</td>
<td>Pm == Pg</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
  bit element1 = ElemP[operand1, e, esize];
  bit element2 = ElemP[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = element1 EOR element2;
  else
    ElemP[result, e, esize] = '0';

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
### 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.

![VEX instruction encoding](image)

**EOR** `<Zdn>`, `<T>`, `<Pg>/M`, `<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><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

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
EOR (immediate)

Bitwise exclusive OR with immediate (unpredicated).

Bitwise exclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction EON.

$$\text{EOR } <\text{Zdn}>, <\text{T}>, <\text{Zdn}>, <\text{T}>, \#<\text{const}>$$

```plaintext
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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV 64;
bits(VL) operand = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(64) element1 = Elem[operand, e, 64];
  Elem[result, e, 64] = element1 EOR imm;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
0 0 0 0 0 1 0 0 1 0 1  Zm  |  0 0 1 1 0 0  |  Zn  |  Zd
```

\[ \text{EOR } <Zd>.D, <Zn>.D, <Zm>.D \]

if \! \texttt{HaveSVE}() then UNDEFINED;
integer n = \texttt{UInt}(Zn);
integer m = \texttt{UInt}(Zm);
integer d = \texttt{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**

\[ \text{CheckSVEEnabled}(); \]
\[ \text{bits}(VL) \ \text{operand1} = Z[n]; \]
\[ \text{bits}(VL) \ \text{operand2} = Z[m]; \]
\[ Z[d] = \text{operand1 EOR operand2}; \]

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>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>

EORV <V><d>, <Pg>, <Zn>.

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;
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 first source vector. This instruction is unpredicated.

An index that is greater than or equal to the vector length in bytes is treated as zero, leaving the destination and first source vector unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 0 0 1 imm8h 0 0 0 imm8l Zm Zdn</td>
</tr>
</tbody>
</table>

```
```

```c
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer position = UInt(imm8h:imm8l);
```

**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.
- `<imm>` Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8h:imm8l" fields.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
if position >= elements then
    position = 0;
position = position << 3;
bits(VL+2) concat = operand2 : operand1;
result = concat<position+VL-1:position>;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point absolute difference (predicated).

Compute the absolute difference of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

FABD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPAbs(FPSub(element1, element2, FPCR[]));
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FABS

Floating-point absolute value (predicated).

Take the absolute value of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This clears the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|------------------------|
| 0 0 0 0 0 1 0 0 | size | 0 1 1 1 0 0 | 1 0 1 | Pg | Zn | Zd |

FABS <Zd>,<T>, <Pg>/M, <Zn>,<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>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] = FPAbs(element);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FAC<cc>**

Floating-point absolute compare vectors.

Compare active absolute values of floating-point elements in the first source vector with corresponding absolute values of elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

The <cc> symbol specifies one of the standard ARM condition codes: GE, GT, LE, or LT.

This instruction is used by the pseudo-instructions FACLE and FACLT.

It has encodings from 2 classes: Greater than and Greater than or equal

### Greater than

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 1 0 0 0 1 0 1 0 0

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;

### Greater than or equal

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 1 0 0 0 0 1 1 0 1 0 0 0 1 0 1 0 0 0 1 0 0

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;

### Assembler Symbols

- **<Pd>** is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<T>** is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Zm>** is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(PL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        case op of
            when Cmp_GE res = FPCompareGE(FPAbs(element1), FPAbs(element2), FPCR[]);
            when Cmp_GT res = FPCompareGT(FPAbs(element1), FPAbs(element2), FPCR[]);
        ElemP[result, e, esize] = if res then '1' else '0';
        else
            ElemP[result, e, esize] = '0';

P[d] = result;
FACLE

Floating-point absolute compare less than or equal.

Compare active absolute values of floating-point elements in the first source vector being less than or equal to corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FAC<cc>. This means:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & \text{Zm} & 1 & 1 & 0 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

FACLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in “size”:

\[
\begin{array}{c|c}
\text{size} & \text{<T>} \\
00 & \text{RESERVED} \\
01 & \text{H} \\
10 & \text{S} \\
11 & \text{D} \\
\end{array}
\]

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FAC<cc> gives the operational pseudocode for this instruction.
FACLTI

Floating-point absolute compare less than.

Compare active absolute values of floating-point elements in the first source vector being less than corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FAC<cc>. This means:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1</td>
</tr>
</tbody>
</table>

FACLTI <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FAC<cc> gives the operational pseudocode for this instruction.
FADD (immediate)

Floating-point add immediate (predicated).

Add an immediate to each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | size | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | Pg | 0 | 0 | 0 | 0 | i1 | Zdn |

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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FADD (vectors, predicated)

Floating-point add vector (predicated).

Add active floating-point elements of the second source vector to corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<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>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></td>
<td></td>
</tr>
</tbody>
</table>

FADD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits<P> mask = P[g];
bits<VL> operand1 = Z[dn];
bits<VL> operand2 = Z[m];
bits<VL> result;
for e = 0 to elements-1
   bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
   if ElemP[mask, e, esize] == '1' then
      Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
   else
      Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicate, or be predicate using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FADD (vectors, unpredicated)

Floating-point add vector (unpredicated).

Add all floating-point elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 1 1 0 0 1 0 1 | size             | Zm               | Zn               |
|                  |                  |                  |                  |

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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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.

FADDA <V><dn>, <Pg>, <V><dn>, <Zm>.

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in &quot;size&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;dn&gt;</th>
<th>Is the number [0-31] of the source and destination SIMD&amp;FP register, encoded in the &quot;Vdn&quot; field.</th>
</tr>
</thead>
<tbody>
<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;Zm&gt;</td>
<td>Is the name of the source scalable vector register, encoded in the &quot;Zm&quot; field.</td>
</tr>
<tr>
<td>&lt;T&gt;</td>
<td>Is the size specifier, encoded in &quot;size&quot;:</td>
</tr>
<tr>
<td>-----</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>size</td>
<td>&lt;T&gt;</td>
</tr>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = V[dn];
bits(VL) operand2 = Z[m];
bits(esize) result = operand1;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem(operand2, e, esize);
    result = FPAdd(result, element, FPCR[]);
V[dn] = result;
FADDV

Floating-point add recursive reduction to scalar.

Floating-point add horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +0.0.

```
 0 1 1 0 0 1 0 1  size  0 0 0 0 0 0 0 0 1  Pg  Zn  Vd
```

FADDV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) identity = FPZero('0');

V[d] = ReducePredicated(ReduceOp_FADD, operand, mask, identity);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
| 31 30 29 28 27 26 25 24 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 | rot | Pg | Zm | Zdn |
| 0 1 1 0 0 1 0 0 | 0 0 0 0 0 | 1 0 0 | 0 |
```

**FCADD** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 0 << 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":
  - size: 00 RESERVED, 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.
- `<const>` is the const specifier, encoded in "rot":
  - rot: 0 #90, 1 #270
Operation

CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
bits(VL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for p = 0 to pairs-1
  acc_r = Elem[operand1, 2 * p + 0, esize];
  acc_i = Elem[operand1, 2 * p + 1, esize];
  elt2_r = Elem[operand2, 2 * p + 0, esize];
  elt2_i = Elem[operand2, 2 * p + 1, esize];
  if ElemP[mask, 2 * p + 0, esize] == '1' then
    if sub_i then elt2_i = FPNeg(elt2_i);
    acc_r = FPAdd(acc_r, elt2_i, FPCR[]);
  if ElemP[mask, 2 * p + 1, esize] == '1' then
    if sub_r then elt2_r = FPNeg(elt2_r);
    acc_i = FPAdd(acc_i, elt2_r, FPCR[]);
  Elem[result, 2 * p + 0, esize] = acc_r;
  Elem[result, 2 * p + 1, esize] = acc_i;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

```
<table>
<thead>
<tr>
<th>0 1 1 0 0 1 0 1</th>
<th>0 1 0 0 1 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>lt</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>0 1 1 0 0 1 0 1</th>
<th>0 1 0 0 0 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>lt</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>0 1 1 0 0 1 0 1</th>
<th>0 1 0 0 0 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>lt</td>
</tr>
</tbody>
</table>
```

FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
Less than

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|
| 0 1 1 0 0 1 0 1 | size           | 0 1 0 0 0 1 0 1|

```
Pg | Zn | 0 | Pd
```

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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|
| 0 1 1 0 0 1 0 1 | size           | 0 1 0 0 0 1 0 1|

```
Pg | Zn | 1 | Pd
```

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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|
| 0 1 1 0 0 1 0 1 | size           | 0 1 0 0 1 1 0 0|

```
Pg | Zn | 0 | Pd
```

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

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
P[d] = result;
**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  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Zm | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg |

cmph cmpl
```

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  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Zm | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg |

cmph cmpl
```

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  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Zm | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg | Zn | Pg |

cmph cmpl
```
FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;

Not equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|---|---|---|---|---|
|                                              | Zm | 1 | 1 | 0 | 1 | 0 |
|                                              | cmph | | | | | |

FCMNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;

Unordered

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|---|---|---|---|---|
|                                              | Zm | 1 | 1 | 0 | 1 | 0 |
|                                              | cmpl | | | | | |

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_LT res = FPCompareLT(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 | size | 0 | Zm | 0 | rot | Pg | Zn | Zda |

**FCMLA** `<Zda>`, `<T>`, `<Pg>/M`, `<Zn>`, `<Zm>`, `<const`

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer e = 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>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESERVED</td>
<td>H</td>
<td>S</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>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#0</td>
<td>#90</td>
<td>#180</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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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**

```assembly
```

```c
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**

```assembly
FCMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>
```

```c
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);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(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 * s + sel_a, esize];
    elt2_b = Elem[operand2, 2 * s + sel_b, esize];
    if neg_r then elt2_a = FPNeg(elt2_a);
    if neg_i then elt2_b = FPNeg(elt2_b);
    addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;
Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FCMLE (vectors)**

Floating-point compare less than or equal to vector.

Compare active floating-point elements in the first source vector being less than or equal to corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of **FCM<cc> (vectors)**. This means:

- The encodings in this description are named to match the encodings of **FCM<cc> (vectors)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **FCM<cc> (vectors)** gives the operational pseudocode for this instruction.

<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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

`cmph` `cmpl`

**FCMLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>**

is equivalent to

**FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>**

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of **FCM<cc> (vectors)** gives the operational pseudocode for this instruction.
FCMLT (vectors)

Floating-point compare less than vector.

Compare active floating-point elements in the first source vector being less than corresponding elements in the second source vector; and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FCM<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>size</td>
<td>0</td>
<td></td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Zn</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>
</tr>
</tbody>
</table>

FCMLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FCMGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- `<Pd>` is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FCM<cc> (vectors) gives the operational pseudocode for this 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).

```
| 0 0 0 0 0 0 1 0 1 | 0 1 | 1 1 0 | imm8 | Zd |
```

FCPY <Zd>,<T>, <Pg>/M, #<const>

```java
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
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>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressable as ±n×16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) result = Z[d];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = imm;
Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FCVT

Floating-point convert precision (predicated).

Convert the size and precision of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.


Half-precision to single-precision

\[
\begin{array}{c|c|c}
0 & 1 & 1 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
\end{array}
\]

```
FCVT <Zd>.S, <Pg>/M, <Zn>.H
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;

Half-precision to double-precision

\[
\begin{array}{c|c|c}
0 & 1 & 1 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
0 & 1 & 1 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
\end{array}
\]

```
FCVT <Zd>.D, <Pg>/M, <Zn>.H
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;

Single-precision to half-precision

\[
\begin{array}{c|c|c}
0 & 1 & 1 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
\end{array}
\]

```
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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 0  1  1  0  0  1  0  1  1  1  0  1  1  1  0  1| Pg                                            | Zn                                            |

```
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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 0  1  1  0  0  1  0  1  1  0  0  1  0  0  1  0  1| Pg                                            | Zn                                            |

```
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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 0  1  1  0  0  1  0  1  1  0  1  0  1  0  1| Pg                                            | Zn                                            |

```
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

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR[]);
        Elem[result, e, esize] = ZeroExtend(res);

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FCVTZS

Floating-point convert to signed integer, rounding toward zero (predicated).

Convert to the signed integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are sign-extended to fill each destination element.

It has encodings from 7 classes: **Half-precision to 16-bit**, **Half-precision to 32-bit**, **Half-precision to 64-bit**, **Single-precision to 32-bit**, **Single-precision to 64-bit**, **Double-precision to 32-bit** and **Double-precision to 64-bit**

**Half-precision to 16-bit**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
```

FCVTZS <Zd>.H, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

**Half-precision to 32-bit**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 0 1 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1
```

FCVTZS <Zd>.S, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

**Half-precision to 64-bit**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 0 1 1 0 1 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1
```

FCVTZS
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  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | Pg | Zn | Zd |

int_U

FCVTZS <Zd>.S, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | Pg | Zn | Zd |

int_U

FCVTZS <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d =UInt(Zd);
integer s_esize = 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  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | Pg | Zn | Zd |

int_U

FCVTZS <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
Double-precision to 64-bit

```
0 1 1 0 0 1 1 1 0 1 1 1 0 1 0 1 | Pg | Zn | Zd

int_U

FCVTZS <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 = 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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FCVTZU

Floating-point convert to unsigned integer, rounding toward zero (predicated).

Convert to the unsigned integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: Half-precision to 16-bit, Half-precision to 32-bit, Half-precision to 64-bit, Single-precision to 32-bit, Single-precision to 64-bit, Double-precision to 32-bit and Double-precision to 64-bit

Half-precision to 16-bit

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & \text{Pg} & \text{Zn} & \text{Zd} \\
\end{array}
\]

int_U

FCVTZU \text{<Zd>}.H, \text{<Pg>}/M, \text{<Zn>}.H

if !\text{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

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & \text{Pg} & \text{Zn} & \text{Zd} \\
\end{array}
\]

int_U

FCVTZU \text{<Zd>}.S, \text{<Pg>}/M, \text{<Zn>}.H

if !\text{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

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & \text{Pg} & \text{Zn} & \text{Zd} \\
\end{array}
\]

int_U
FCVTZU <Zd>.D, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 32-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0
int_U

FCVTZU <Zd>.S, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 64-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0
int_U

FCVTZU <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 32-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 1 1 0 1 1 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0
int_U

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

| 31 30 29 28 27 26 25 24 23 22 21 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 | 1 | 1 | 1 | 0 | 1 | Pg | Zn | Zd |

int_U

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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

### 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] = FPDiv(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FDIVR**

Floating-point reversed divide by vector (predicated).

Reversed divide active floating-point elements of the second source vector by corresponding floating-point elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 1   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | 0   | Pg  | Zm  | Zdn |

**FDIVR** `<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>`

```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);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPDiv(element2, element1, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FDUP

Broadcast 8-bit floating-point immediate to vector elements (unpredicated).

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias FMOV (immediate, unpredicated).

| 31 30 29 28 27 26 25 24 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                           | imm8            | Zd              |
| 0 0 0 1 0 0 1 0 1              | 1 1 1 0 0 1 1 1 0 |                |

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 \((\text{src}<16:6>)\), 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 \((\text{src}<13:6>)\), 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 \((\text{src}<9:5>)\), 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.

FEXPA \(<Zd>\,<T>\), \(<Zn>\,<T>

\[
\begin{array}{c|c|ccc|ccc|cccc|c|c|}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & Zn & Zd
\end{array}
\]

\[
\begin{array}{cccc}
31 & 30 & 29 & 28 \\
27 & 26 & 25 & 24 \\
23 & 22 & 21 & 20 \\
19 & 18 & 17 & 16 \\
15 & 14 & 13 & 12 \\
11 & 10 & 9 & 8 \\
7 & 6 & 5 & 4 \\
3 & 2 & 1 & 0 \\
\end{array}
\]

Assembler Symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the “Zd” field.

\(<T>\) Is the size specifier, encoded in “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

\[
\begin{align*}
\text{integer elements} &= \text{VL} \div \text{esize} ; \\
\text{bits}(\text{VL}) \text{ operand} &= Z[n] ; \\
\text{bits}(\text{VL}) \text{ result} ; \\
\text{for e = 0 to elements-1} \\
&\quad \text{bits(esize)} \text{ element} = \text{Elem}(\text{operand, e, esize}) ; \\
&\quad \text{Elem}(\text{result, e, esize}) = \text{FPExpA}(\text{element}) ; \\
Z[d] &= \text{result} ;
\end{align*}
\]
**FMAD**

Floating-point fused multiply-add vectors (predicated), writing multiplicand \([Zdn = Za + Zdn \times Zm]\).

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) 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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>g</td>
<td>Z</td>
<td>m</td>
<td>Z</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**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”:
  - size | <T>
  - 00 | RESERVED
  - 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.
- `<Za>`: Is the name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[a];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
bits(esize) element3 = Elem[operand3, e, esize];
  if ElemP[mask, e, esize] == '1' then
    if op1_neg then element1 = FPNeg(element1);  
    if op3_neg then element3 = FPNeg(element3);  
      Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**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.

```
size 0 1 1 0 1 0 1
Pg 0 0 0 0 0 0 0
i1 0 0 0 1 1 0
Zdn
```

FMAX <Zdn>., <T>., <Pg>/M, <Zdn>.<T>., <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

### Assembler Symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":
  ```
  size  <T>
  00  RESERVED
  01  H
  10  S
  11  D
  ```
- `<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":
  ```
  i1  <const>
  0  #0.0
  1  #1.0
  ```

### Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMax(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMAX (vectors)**

Floating-point maximum (predicated).

Determine the maximum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If either element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 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 | 0 | Pg | Zm | Zdn |

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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAXNM (Immediate)

Floating-point maximum number with immediate (predicated).

Determine the maximum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is the immediate. Inactive elements in the destination vector register remain unmodified.

```
0 1 1 0 1 0 1 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 1 0
```

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);
bite(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];
bite(VL) operand1 = Z[dn];
bite(VL) result;
for e = 0 to elements-1
    bite(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMaxNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAXNM (vectors)

Floating-point maximum number (predicated).

Determine the maximum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is NaN then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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  | 1  | 0  | 0  | 0  | 1  | Pg  | Zn  | Vd  |

FMAXNMV <V><d>, <Pg>, <Zn><T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<V> is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> is the name of the source scalable vector register, encoded in the "Zn" field.

<T> is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) identity = FPDefaultNaN();

V[d] = ReducePredicated(ReduceOp_FMAXNUM, operand, mask, identity);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
0 1 1 0 0 1 0 1 | size | 0 0 0 1 | 1 | 0 0 0 1 | Pg | Zn | Vd
```

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><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
- `< Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `< Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

```
CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) identity = FPInfinity('1');
V[d] = ReducePredicated(ReduceOp_FMAX, operand, mask, identity);
```

Copyright © 2010-2020 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.

```
FMIN <Zdn>.<T>, <Pg >/M, <Zdn>.<T>, <const>
```  

if !HaveSVE() then UNDEFINED;  
if size == '00' then UNDEFINED;  
integer esize = 8 << UInt(size);  
integer g = UInt(Pg);  
integer dn = UInt(Zdn);  
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

Assembler Symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<const>` Is the floating-point immediate value, encoded in “i1”:
  
<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMin(element1, imm, FPCR[]);  
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```  

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMIN (vectors)**

Floating-point minimum (predicated).

Determine the minimum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If either element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.

```
<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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>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>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>`

```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);
```

**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   RESERVED
  01   H
  10   S
  11   D
  ```
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMIN(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMINNM (immediate)

Floating-point minimum number with immediate (predicated).

Determine the minimum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is the immediate. Inactive elements in the destination vector register remain unmodified.

|    |    |    |    | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |    |    |    | Pg | 0 | 0 | 0 | 0 | i1 |    | Zdn |
|----|----|----|----|---|---|---|---|---|---|---|---|---|----|----|----|---|---|---|---|---|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMinNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMINNM (vectors)

Floating-point minimum number (predicated).

Determine the minimum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is NaN then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P[1]) 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] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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 1 | Pg              | Zn              | Vd              |

FMINNMV <V><d>, <Pg>, <Zn>.

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(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_FMINNUM, operand, mask, identity);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINV

Floating-point minimum recursive reduction to scalar.

Floating-point minimum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the
result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +Infinity.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 0 0 1 0 1 | size | 0 0 0 | 1 1 1 0 0 1 | Pg | Zn | Vd |

FMINV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(<PL>) mask = P[g];
bits(<VL>) operand = Z[n];
bits(esize) identity = FPInfinity('0');
V[d] = ReducePredicated(ReduceOp_FMIN, operand, mask, identity);
FMLA (vectors)

Floating-point fused multiply-add vectors (predicated), writing addend \([Zda = Zda + Zn \times Zm]\).

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 results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | size | 1 | Zm | 0 | 0 | 0 | Pg | Zn | Zda |

FMLA \(<Zda>\).<T>, \(<Pg>/M, <Zn>.<T>, <Zm>.<T>\>

if !\(\text{HaveSVE}()\) then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer 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

\(\text{CheckSVEEnabled}()\);
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 = \(\text{Elem}\) [operand1, e, esize];
    bits(esize) element2 = \(\text{Elem}\) [operand2, e, esize];
    bits(esize) element3 = \(\text{Elem}\) [operand3, e, esize];
    if \(\text{ElemP}[mask, e, esize] == '1'\) then
        if op1_neg then element1 = \(\text{FPNeg}\) [element1];
        if op3_neg then element3 = \(\text{FPNeg}\) [element3];
        \(\text{Elem}\) [result, e, esize] = \(\text{FPMulAdd}\) [element3, element1, element2, FPCR[]];
    else
        \(\text{Elem}\) [result, e, esize] = element3;
    end
\(Z[da] = \) result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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

```
0 1 1 0 1 0 0 1 | i3h:1 | i3l  | Zm 0 0 0 0 0 0 | Zn | Zda

```

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 = FALSE;
boolean op3_neg = FALSE;

### Single-precision

```
0 1 1 0 1 0 0 1 | 0 1 | i2  | Zm 0 0 0 0 0 0 | Zn | Zda

FMLA <Zda>.S, <Zn>.S, <Zm>[<imm>]
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

### Double-precision

```
0 1 1 0 1 0 0 1 | 1 1 | i1  | Zm 0 0 0 0 0 0 | Zn | Zda

FMLA <Zda>.D, <Zn>.D, <Zm>[<imm>]
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
Assembler Symbols

<Zda>       Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn>        Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>        For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
             For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
<imm>       For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
             For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.
             For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLS (vectors)

Floating-point fused multiply-subtract vectors (predicated), writing addend \([Zda = Zda + -Zn \times Zm]\).

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | | size | 1 | | Zm | 0 | 0 | 1 | | Pg | 0 | 0 | 1 | 0 | | Zn | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |

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(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements - 1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  bits(esize) element3 = Elem[operand3, e, esize];
  if ElemP[mask, e, esize] == '1' then
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element3;
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLS (indexed)

Floating-point fused multiply-subtract by indexed elements (Zda = Zda + -Zn * Zm[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 | 0  | 0  | 0  | 1  | Zda |


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 | 0  | 1  | i2 | Zm | 0  | 0  | 0  | 0  | 0  | 1  | Zn | Zda |

FMLS <Zda>.S, <Zn>.S, <Zm>[<imm>]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

Double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0 | 1  | 1  | i1 | Zm | 0  | 0  | 0  | 0  | 0  | 1  | Zn | Zda |

FMLS <Zda>.D, <Zn>.D, <Zm>[<imm>]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.
For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the “i1" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, s, esize];
  bits(esize) element3 = Elem[result, e, esize];
  if op1_neg then element1 = FPNeg(element1);
  if op3_neg then element3 = FPNeg(element3);
  Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMOV (zero, predicated)

Move floating-point +0.0 to vector elements (predicated).

Move floating-point constant +0.0 to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is a pseudo-instruction of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

FMOV <Zd>.<T>, <Pg>/M, #0.0

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #0

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>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

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMOV (zero, unpredicated)**

Move floating-point +0.0 to vector elements (unpredicated).

Unconditionally broadcast the floating-point constant +0.0 into each element of the destination vector. This instruction is unpredicated.

This is a pseudo-instruction of **DUP (immediate)**. This means:

- The encodings in this description are named to match the encodings of **DUP (immediate)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **DUP (immediate)** gives the operational pseudocode for this instruction.

```
FMOV <Zd>.<T>, #0.0
```

is equivalent to

```
DUP <Zd>.<T>, #0
```

**Assembler Symbols**

- **<Zd>** is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

The description of **DUP (immediate)** gives the operational pseudocode for this instruction.

---

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (immediate, predicated)

Move 8-bit floating-point immediate to vector elements (predicated).

Move a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of FCPY. This means:

- The encodings in this description are named to match the encodings of FCPY.
- The description of FCPY gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

FMOV <Zd>.<T>, <Pg>/M, #<const>

is equivalent to

FCPY <Zd>.<T>, <Pg>/M, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressable as ±n×16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

Operation

The description of FCPY gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMOV (immediate, unpredicated)

Move 8-bit floating-point immediate to vector elements (unpredicated).

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This is an alias of FDUP. This means:

- The encodings in this description are named to match the encodings of FDUP.
- The description of FDUP gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 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 |

FMOV <Zd>,<T>, #<const>

is equivalent to

FDUP <Zd>,<T>, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> is a floating-point immediate value expressable as \( \pm n \times 16 \times 2^r \), where \( n \) and \( r \) are integers such that \( 16 \leq n \leq 31 \) and \( -3 \leq r \leq 4 \), i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of FDUP gives the operational pseudocode for this instruction.
FMSB

Floating-point fused multiply-subtract vectors (predicated), writing multiplicand \(Z_{dn} = Z_a + Z_{dn} \times Z_m\).

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

FMSB \(<Zdn>.<T>, \ <Pg>/M, \ <Zm>.<T>, \ <Za>.<T>\)

if \(!\texttt{HaveSVE}()\) 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

\(\texttt{CheckSVEEnabled}();\)
integer elements = \(VL\) DIV esize;
bits(PL) mask = \(P\)[g];
bits(VL) operand1 = \(Z\)[dn];
bits(VL) operand2 = \(Z\)[m];
bits(VL) operand3 = \(Z\)[a];
bits(VL) result;
for \(e = 0\) to elements-1
    bits(esize) element1 = \(\texttt{Elem}[\texttt{operand1}, \ e, \ \text{esize}];\)
    bits(esize) element2 = \(\texttt{Elem}[\texttt{operand2}, \ e, \ \text{esize}];\)
    bits(esize) element3 = \(\texttt{Elem}[\texttt{operand3}, \ e, \ \text{esize}];\)
    if \(\texttt{ElemP}[\texttt{mask}, \ e, \ \text{esize}] == '1'\) then
        if op1_neg then element1 = \(\texttt{FPNeg}(\text{element1});\)
        if op3_neg then element3 = \(\texttt{FPNeg}(\text{element3});\)
        \(\texttt{Elem}[\text{result}, \ e, \ \text{esize}] = \texttt{FPMulAdd}(\text{element3}, \ \text{element1}, \ \text{element2}, \ \text{FPCR[]});\)
    else
        \(\texttt{Elem}[\text{result}, \ e, \ \text{esize}] = \text{element1};\)
\(Z[dn] = \text{result};\)
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|-----------------------------------------------|
| 0  1  1  1  0  0  1  0  1  0  1  0  1  0  0 | size  0  1  1  0  1  0  1  0  0 | Pg  0  0  0  0 | i1 | Zdn |

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);
bites(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;
bites(VL) 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 ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMul(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMUL (vectors, predicated)**

Floating-point multiply vectors (predicated).

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMUL (vectors, unpredicated)**

Floating-point multiply vectors (unpredicated).

Multiply all elements of the first source vector by corresponding floating-point elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0</td>
<td>0 0 0 0 1 0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

**Asmmler 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**

```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] = FPMul(element1, element2, FPCR[]);

Z[d] = result;
```

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMUL (indexed)

Floating-point multiply by indexed elements.

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision and Double-precision

Half-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | Zm | 0  | 0  | 1  | 0  | 0  | 0  | Zn  |    |    |    |    | Zd  |


if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Single-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | i2 | Zm | 0  | 0  | 1  | 0  | 0  | 0  | Zn  |    |    |    |    | Zd  |

size<1>|size<0>

FMUL <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | i1 | Zm | 0  | 0  | 1  | 0  | 0  | 0  | Zn  |    |    |    |    | Zd  |

size<1>|size<0>


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**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
Z[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMULX**

Floating-point multiply-extended vectors (predicated).

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector except that $\infty \times 0.0$ gives 2.0 instead of NaN, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

The instruction can be used with FRECPX to safely convert arbitrary elements in mathematical vector space to UNIT VECTORS or DIRECTION VECTORS with length 1.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 | size | 0 0 | 1 0 1 | 0 | 1 0 0 | Pg | Zm | Zdn
```

**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] = 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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

### 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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P(g);
bits(VL) operand = Z(n);
bits(VL) result = Z(d);
for e = 0 to elements-1
    bits(esize) element = Elem(operand, e, esize);
    if ElemP(mask, e, esize) == '1' then
        Elem(result, e, esize) = FPNeg(element);
Z[d] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.
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.

FNMAD symbol meanings:

- <Zdn>: Name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <T>: Size specifier, encoded in "size":
  - 00: RESERVED
  - 01: H
  - 10: S
  - 11: D
- <Pg>: Name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zm>: Name of the second source scalable vector register, encoded in the "Zm" field.
- <Za>: Name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

```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);
integer a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[a];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
bits(esize) element3 = Elem[operand3, e, esize];
  if ElemP[mask, e, esize] == '1' then
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point negated fused multiply-add vectors (predicated), writing addend \( \text{Zda} = -\text{Zda} + -\text{Zn} \times \text{Zm} \).

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

If \texttt{CheckSVEEnabled}() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer g = \texttt{UInt}(Pg);
integer n = \texttt{UInt}(Zn);
integer m = \texttt{UInt}(Zm);
integer da = \texttt{UInt}(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;

Assembler Symbols

\(<\text{Zda}>\) Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

\(<\text{T}>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;\text{T}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zn}>\) Is the name of the 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

\texttt{CheckSVEEnabled}();
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(\texttt{PL}) mask = \texttt{P}[g];
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
  bits(esize) element1 = \texttt{Elem}[operand1, e, esize];
  bits(esize) element2 = \texttt{Elem}[operand2, e, esize];
  bits(esize) element3 = \texttt{Elem}[operand3, e, esize];
  if \texttt{ElemP}[mask, e, esize] == '1' then
    if op1_neg then element1 = \texttt{FPNeg}(element1);
    if op3_neg then element3 = \texttt{FPNeg}(element3);
    \texttt{Elem}[result, e, esize] = \texttt{FPMulAdd}(element3, element1, element2, \texttt{FPCR}[]);
  else
    \texttt{Elem}[result, e, esize] = element3;
\texttt{Z}[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FNMLS

Floating-point negated fused multiply-subtract vectors (predicated), writing addend \([Z_{da} = -Z_{da} + Z_{n} \times Z_{m}]\).

Multiply the corresponding active floating-point elements of the first and second source vectors and 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.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & Z_{m} & 0 & 1 & 1 & P_{g} & Z_{n} & Z_{da} & N & o & p
\end{array}
\]

**Assembler Symbols**

- **<Z_{da}>** Is the name of the third source and destination scalable vector register, encoded in the “Z_{da}” field.
- **<T>** Is the 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>

- **<P_{g}>** Is the name of the governing scalable predicate register P0-P7, encoded in the "P_{g}" field.
- **<Z_{n}>** Is the name of the first source scalable vector register, encoded in the "Z_{n}" field.
- **<Z_{m}>** Is the name of the second source scalable vector register, encoded in the "Z_{m}" field.

**Operation**

\[
\text{CheckSVEEnabled}();
\]

integer elements = \(\text{VL} \div \text{esize}\);
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
\[
\text{bits(esize)} \text{ element1} = \text{Elem}[\text{operand1}, e, \text{esize}];
\text{bits(esize)} \text{ element2} = \text{Elem}[\text{operand2}, e, \text{esize}];
\text{bits(esize)} \text{ element3} = \text{Elem}[\text{operand3}, e, \text{esize}];
\]

if \(\text{ElemP}[\text{mask}, e, \text{esize}] == '1'\) then
\[
\text{if op1_neg then element1} = \text{FPNeg}(\text{element1});
\text{if op3_neg then element3} = \text{FPNeg}(\text{element3});
\]
else
\[
\text{Elem}[\text{result}, e, \text{esize}] = \text{FPMulAdd}(\text{element3}, \text{element1}, \text{element2}, \text{FPCR}[]);
\]
\[
Z[da] = \text{result};
\]
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand \[Z_{dn} = -Z_{a} + Z_{dn} \times Z_{m}\].

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

\[\begin{array}{ccccccccccccccccccc}
\end{array}\]

\[\begin{array}{cccc}
0 & 1 & 1 & 0
\end{array}\] | Za | 1 | 1 | Pg | Zm | Zdn
\[\text{N op}\]

FNMSB \(<\text{Zdn}>.<\text{T}>, <\text{Pg}>/M, <\text{Zm}>.<\text{T}>, <\text{Za}>.<\text{T}>\)

\[
\begin{align*}
\text{if } & \text{HaveSVE}() \text{ then UNDEFINED;} \\
\text{if } & \text{size} == \text{'00'} \text{ then UNDEFINED;} \\
\text{integer } & \text{esize} = 8 \ll \text{UInt}(\text{size}); \\
\text{integer } & g = \text{UInt}(\text{Pg}); \\
\text{integer } & dn = \text{UInt}(\text{Zdn}); \\
\text{integer } & m = \text{UInt}(\text{Zm}); \\
\text{integer } & a = \text{UInt}(\text{Za}); \\
\text{boolean } & \text{op1_neg} = \text{FALSE}; \\
\text{boolean } & \text{op3_neg} = \text{TRUE};
\end{align*}
\]

**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”:

\[
\begin{array}{c|c}
\text{size} & \text{<T>} \\
\hline
00 & \text{RESERVED} \\
01 & H \\
10 & S \\
11 & D \\
\end{array}
\]

\(<\text{Pg}>\) 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.

\(<\text{Za}>\) Is the name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

\[
\text{CheckSVEEnabled}();
\text{integer elements} = \text{VL} \div \text{esize};
\text{bits(PL)} \text{ mask} = \text{P}[g];
\text{bits(VL)} \text{ operand1} = \text{Z}[dn];
\text{bits(VL)} \text{ operand2} = \text{Z}[m];
\text{bits(VL)} \text{ operand3} = \text{Z}[a];
\text{bits(VL)} \text{ result};
\text{for } e = 0 \text{ to elements-1}
\text{bits(esize)} \text{ element1} = \text{Elem}[\text{operand1}, e, \text{esize}];
\text{bits(esize)} \text{ element2} = \text{Elem}[\text{operand2}, e, \text{esize}];
\text{bits(esize)} \text{ element3} = \text{Elem}[\text{operand3}, e, \text{esize}];
\text{if } \text{ElemP}[(\text{mask}, e, \text{esize}) == \text{'}1\text{'}] \text{ then}
\text{if } \text{op1neg} \text{ then element1} = \text{FPNeg}()[\text{element1}];
\text{if } \text{op3neg} \text{ then element3} = \text{FPNeg}()[\text{element3}];
\text{Element}[(\text{result}, e, \text{esize}) = \text{FPMulAdd}()[\text{element3}, \text{element1}, \text{element2}, \text{FPCR}[]]);
\text{else}
\text{Element}[(\text{result}, e, \text{esize}) = \text{element1}];
\text{Z}[dn] = \text{result};
\]

FNMSB

Page 1659
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FRECPE

Floating-point reciprocal estimate (unpredicated).

Find the approximate reciprocal of each floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 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 |

FRECPE  \(<Zd>\).<T>,  \(<Zn>\).<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

\(<Zd>\)  Is the name of the destination scalable vector register, encoded in the "Zd" field.
\(<T>\)  Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Zn>\)  Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

\(\text{CheckSVEEnabled}()\);
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] = FPRecipEstimate(element, FPCR[]);
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRECPS

Floating-point reciprocal step (unpredicated).

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 2.0 without intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal of a vector of floating-point values.

| 31 30 29 28 27 26 25 24 23 22 21 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 | 0 | Zn | Zd |

FRECPS <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] = FPRecipStepFused(element1, element2);
Z[d] = result;
**FRECPX**

Floating-point reciprocal exponent (predicated).

Invert the exponent and zero the fractional part of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

The result of this instruction can be used with FMULX to convert arbitrary elements in mathematical vector space to "unit vectors" or "direction vectors" of length 1.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 0 1 1 0 0 1 0 1 | size   | 0 0 1 1 0 0 1 0 | Pg      | Zn      | Zd      |

FRECPX <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] = FPRrecpX(element, FPCR[]);
Z[d] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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 |
|-----------------------------|-----------------------------|-----------------------------|
| size | Pg | Zn | Zd |
| 0 1 1 0 0 1 0 1 | 0 0 0 | 1 1 1 1 1 0 1 | 0 0 0 |

FRINTI <Zd>, <Pg>/M, <Zn>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

Current mode signalling inexact

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|
| size | Pg | Zn | Zd |
| 0 1 1 0 0 1 0 1 | 0 0 0 | 1 1 0 1 0 1 | 1 0 |

FRINTX <Zd>, <Pg>/M, <Zn>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

Nearest with ties to away

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|
| size | Pg | Zn | Zd |
| 0 1 1 0 0 1 0 1 | 0 0 0 | 1 0 0 1 0 1 | 0 1 |

FRINTA <Zd>, <Pg>/M, <Zn>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEAWAY;
Nearest with ties to even

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 1 0 0 1 0 1 | size 0 0 0 0 0 1 0 1 | Pg  Zn  Zd
```

FRINTN <Zd>.<T>, <Pg>/M, <Zn>.<T>

```java
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEEVEN;
```

Toward zero

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 1 0 0 1 0 1 | size 0 0 0 0 1 1 0 1 | Pg  Zn  Zd
```

FRINTZ <Zd>.<T>, <Pg>/M, <Zn>.<T>

```java
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Toward minus infinity

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 1 0 0 1 0 1 | size 0 0 0 0 1 0 1 0 | Pg  Zn  Zd
```

FRINTM <Zd>.<T>, <Pg>/M, <Zn>.<T>

```java
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_NEGINF;
```

Toward plus infinity

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 1 0 0 1 0 1 | size 0 0 0 0 1 1 0 1 | Pg  Zn  Zd
```

FRINTP <Zd>.<T>, <Pg>/M, <Zn>.<T>

```java
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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRSQRTE

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 |
|---------------------------------|---------------|--------|
| 0 1 1 0 0 1 0 1               | size          | 0 0 1 1 1 1 0 0 |
| Zn                             | Zd            |

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":

<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] = FPRSqrtEstimate(element, FPCR[]);
Z[d] = result;
Floating-point reciprocal square root step (unpredicated).

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 3.0 and divide the results by 2.0 without any intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal square root of a vector of floating-point values.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 | size 0 | Zm 0 0 0 1 1 1 | Zn | Zd
```

FRSQRTS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
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.

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 ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPScale(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |

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 ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPSqrt(element, FPCR[]);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUB (immediate)

Floating-point subtract immediate (predicated).

Subtract an immediate from each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUB (vectors, predicated)

Floating-point subtract vectors (predicated).

Subtract active floating-point elements of the second source vector from corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
0 1 1 0 0 1 0 1 size 0 0 0 0 1 1 0 0 Pg Zm Zdn
```

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":
  - `00` RESERVED
  - `01` H
  - `10` S
  - `11` D
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUB (vectors, predicated)
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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th></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>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</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>
</tr>
</tbody>
</table>

FSUB <Zd>,<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSUB(element1, element2, FPCR[]);
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUBR (immediate)

Floating-point reversed subtract from immediate (predicated).

Reversed subtract from an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | Pg | 0  | 0  | 0  | 0  | i1 | Zdn|
```

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><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<const>` Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th><code>&lt;const&gt;</code></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

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPSub(imm, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUBR (vectors)

Floating-point reversed subtract vectors (predicated).

Reversed subtract active floating-point elements of the first source vector from corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

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

Assembler Code:

```
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 elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
   bits(esize) element1 = Elem[operand1, e, esize];
   bits(esize) element2 = Elem[operand2, e, esize];
   if ElemP[mask, e, esize] == '1' then
      Elem[result, e, esize] = FPSub(element2, element1, FPCR[]);
   else
      Elem[result, e, esize] = element1;
Z[dn] = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FTMAD**

Floating-point trigonometric multiply-add coefficient.

The FTMAD instruction calculates the series terms for either \( \sin(x) \) or \( \cos(x) \), where the argument \( x \) has been adjusted to be in the range \(-\pi/4 < x \leq \pi/4\).

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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

FTMAD <Zdn>.<T>,<Zm>.<T>,<Zm>.<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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FTSMUL**

Floating-point trigonometric starting value.

The FTSMUL instruction calculates the initial value for the FTMAD instruction. The instruction squares each element in the first source vector and then sets the sign bit to a copy of bit 0 of the corresponding element in the second source register, and places the results in the destination vector. This instruction is unpredicated.

To compute \( \sin(x) \) or \( \cos(x) \) the instruction is executed with elements of the first source vector set to \( x \), adjusted to be in the range \(-\pi/4 < x \leq \pi/4\).

The elements of the second source vector hold the corresponding value of the quadrant \( \phi \) number as an integer not a floating-point value. The value \( \phi \) 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  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 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":
  - \(00\) 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.

**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] = FPTriGSMul(element1, element2, FPCR[]);
Z[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \((2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4\).

### 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 = \( Elem[\text{operand1}, e, \text{esize}] \);
  - bits(esize) element2 = \( Elem[\text{operand2}, e, \text{esize}] \);
  - \( Elem[\text{result}, e, \text{esize}] = FPTrigSSel(\text{element1}, \text{element2}) \);
- \( Z[d] \) = result;

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
INCB, INCD, INCH, INCW (scalar)

Increment scalar by multiple of predicate constraint element count.

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword and Word

Byte

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 1 0 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 0 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 0 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>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(64) operand1 = X[dn];

X[dn] = operand1 + (count * imm);
INCD, INCH, INCW (vector)

Increment vector by multiple of predicate constraint element count.

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: Doubleword, Halfword and Word.

**Doubleword**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 1 1 1 1 imm4 1 1 0 0 0 0 pattern Zdn

size<1=size<0> D

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;

**Halfword**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 1 1 1 1 imm4 1 1 0 0 0 0 pattern Zdn

size<1=size<0> D

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;

**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 0 1 0 1 1 imm4 1 1 0 0 0 0 pattern Zdn

size<1=size<0> D

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>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
INCP (scalar)

Increment scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Pm</th>
<th></th>
<th></th>
<th>Rdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

INCP `<Xdn>`, `<Pm>`.<`T>`

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);

Assembler Symbols

`<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

`<Pm>` Is the name of the source scalable predicate register, encoded in the “Pm” field.

`<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><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>

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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
31 30 29 28 27 26 25 24 23 22 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 1 0 0 | 1 0 0 0 0 0 0 0 0 0 0 Pm  Zdn
```

INCP <Zdn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(Pl) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  Elem[result, e, esize] = Elem[operand1, e, esize] + count;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
INDEX (immediates)

Create index starting from and incremented by immediate.

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element.

This instruction is unpredicated.

```
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><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</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;
```
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.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 size | 1 | Rm | 0 1 0 0 1 0 | imm5 | Zd
```

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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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</td>
</tr>
</tbody>
</table>

INDEX <Zd>.<T>, <R><n>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
integer imm = SInt(imm5);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(esize) operand1 = X[n];
integer element1 = SInt(operand1);
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;
INDEX (scalars)

Create index starting from and incremented by general-purpose register.

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operands are general-purpose registers in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 0 0 0 0 1 0 0 | size | 1 | Rm | 0 1 0 0 1 1 | Rn | Zd
```

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":

```
  | size | <T> |
---|------|------|
 00 | B    |
 01 | H    |
 10 | S    |
 11 | D    |
```

<R> Is a width specifier, encoded in "size":

```
  | size | <R> |
---|------|------|
 01 | W    |
 x0 | W    |
 11 | X    |
```

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**INSR (scalar)**

Insert general-purpose register in shifted vector.

Shift the destination vector left by one element, and then place a copy of the least-significant bits of the general-purpose register in element 0 of the destination vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 | size | 1 0 0 1 0 0 0 1 1 1 0 | Rm | Zdn
```

INSR <Zdn>.<T>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Rm);

### Assembler Symbols

- `<Zdn>` is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<R>` is a width specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

- `<m>` is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

### Operation

**CheckSVEEnabled();**

bits(VL) dest = Z[dn];
bits(esize) src = X[m];
Z[dn] = dest<VL-esize-1:0> : src;

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**INSR (SIMD&FP scalar)**

Insert SIMD&FP scalar register in shifted vector.

Shift the destination vector left by one element, and then place a copy of the SIMD&FP scalar register in element 0 of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<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>

**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();
bias(VL) dest = Z[dn];
bias(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. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LASTA (scalar)

Extract element after last to general-purpose register.

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then zero-extend and place the extracted element in the destination general-purpose register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | 1  | 0  | 1  | 0  | Pg | Zn | Rd |

LASTA &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

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<d> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(rsize) result;
integer last = LastActiveElement(mask, esize);
if isBefore then
  if last < 0 then last = elements - 1;
else
  last = last + 1;
  if last >= elements then last = 0;
result = ZeroExtend(Elem[operand, last, esize]);

X[d] = result;

Copyright © 2010-2020 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.

```
0 0 0 0 0 1 0 1 size 1 0 0 0 1 0 1 0 Pg Zn Vd
```

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);
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>1 0 0 0 0 1 1 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Rd</th>
</tr>
</thead>
</table>

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;

Assembler Symbols

\(<R>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;R&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zn>\) Is the name of the source scalable vector register, encoded in the "Zn" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(rsize) result;
integer last = LastActiveElement(mask, esize);
if isBefore then
  if last < 0 then last = elements - 1;
else
  last = last + 1;
if last >= elements then last = 0;
result = ZeroExtend(Elem[operand, last, esize]);
X[d] = result;
**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.

```
31 30 29 28 27 26 25 24 23 22 21 20 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 1 1 1 0 0 Pg Zn Vd
```

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);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1B (vector plus immediate)

Gather load unsigned bytes to vector (immediate index).

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

msz<1>msz<0> U ff

LD1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

msz<1>msz<0> U ff

LD1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
```
LD1B (scalar plus immediate)

Contiguous load unsigned bytes to vector (immediate index).

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element.

8-bit element

```
LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

16-bit element

```
LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```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 = TRUE;
integer offset = SInt(imm4);
```

32-bit element

```
LD1B { <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 = TRUE;
integer offset = SInt(imm4);
```
64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm4</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>
</tr>
</tbody>
</table>

dtype<3:1> dtype<0>

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
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = result;
LD1B (scalar plus scalar)

Contiguous load unsigned bytes to vector (scalar index).

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 0 0 0 0 0 Rm 0 1 0 Pg Rn Zt
dtype<3:1> dtype<0>

LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;

16-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 0 0 0 1 Rm 0 1 0 Pg Rn Zt
dtype<3:1> dtype<0>

LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;

32-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 0 0 1 0 0 Rm 0 1 0 Pg Rn Zt
dtype<3:1> dtype<0>
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

<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>0</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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1> dtype<0>

LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  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;
LD1B (scalar plus vector)

Gather load unsigned bytes to vector (vector index).

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset

### 32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
<td>0 0 xs 0 Zm 0 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff

LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
<td>0 0 xs 0 Zm 0 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff

LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
<td>0 0 xs 0 Zm 1 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff
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 SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
**LD1D (vector plus immediate)**

Gather load doublewords to vector (immediate index).

Gather load of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23</th>
<th>22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

LD1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD1D (scalar plus immediate)**

Contiguous load doublewords to vector (immediate index).

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

| 31 30 29 28 27 26 25 24 23 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 1 1 1 1 1 0 | imm4 | 1 0 1 | Pg | Rn | 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.
- `<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
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);
```

```plaintext
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);
```

```plaintext
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = X[n];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
```

```plaintext
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;
```

```plaintext
Z[t] = result;
```
**LD1D (scalar plus scalar)**

Contiguous load doublewords to vector (scalar index).

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | Rm | 0  | 1  | 0  | Pg | Rn | Zt |

dtype<3:1> dtype<0>

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

\texttt{CheckSVEEnabled();}
integer elements = \texttt{VL} \div \texttt{esize};

\texttt{bits(64) base;}
\texttt{bits(64) addr;}
\texttt{bits(PL) mask = \texttt{P}[g];}
\texttt{bits(VL) result;}
\texttt{bits(msize) data;}
\texttt{bits(64) offset = \texttt{X}[m];}
constant integer mbytes = msize \div 8;

if \texttt{HaveMTEExt()} then \texttt{SetTagCheckedInstruction(TRUE)};

if \texttt{n == 31} then
  if \texttt{LastActiveElement(mask, esize) \geq 0 ||}
     \texttt{ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)} then
    \texttt{CheckSPAinement();}
  base = \texttt{SP[ ];}
else
  base = \texttt{X[n];}

for \texttt{e = 0} to \texttt{elements-1}
  \texttt{addr = base + UInt(offset) \ast mbytes;}
  if \texttt{ElemP(mask, e, esize) == \texttt{'1'}} then
    \texttt{data = Mem[addr, mbytes, AccType_NORMAL];}
    \texttt{Elem[result, e, esize] = Extend(data, esize, unsigned);} 
  else
    \texttt{Elem[result, e, esize] = Zeros();}
  offset = offset + 1;

\texttt{Z[t] = result;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1D (scalar plus vector)

Gather load doublewords to vector (vector index).

Gather load of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

32-bit unpacked scaled offset

```
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>]
```

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)  Page 1717
```
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  | 1  | 0  | 1   | 1   | 0   | Zm | 1   | 1   | 0   | Pg  | Rn  | Zt  |

msz<1>msz<0> U ff

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 HaveMTEEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPA(kind);
    base = SP[];
else
    base = X[n];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1H (vector plus immediate)

Gather load unsigned halfwords to vector (immediate index).

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</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>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>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
<td></td>
<td>U ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
```

```
LD1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### 64-bit element

```
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>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>0</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>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
<td></td>
<td>U ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
```

```
LD1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### Assembler Symbols

- `<Zt>`
  - Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`
  - Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>`
  - Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>`
  - Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL \text{ 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 \text{ DIV} 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset \ast 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 not cause a read from Device
memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | imm4 | 1  | 0  | 1  | Pg  | Rn  | Zt  |

dtype<3:1>dtype<0>
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  | 0  | imm4 | 1  | 0  | 1  | Pg  | Rn  | Zt  |

dtype<3:1>dtype<0>
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  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | imm4 | 1  | 0  | 1  | Pg  | Rn  | Zt  |

dtype<3:1>dtype<0>
LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits<PL> mask = P<g>;
bits<VL> result;
bits<msize> data;
constant integer mbytes = msize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
  Elem[result, e, esize] = Extend(data, esize, unsigned);
else
  Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = result;
**LD1H (scalar plus scalar)**

Contiguous load unsigned halfwords to vector (scalar index).

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element.

### 16-bit element

```
<table>
<thead>
<tr>
<th>31</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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**LD1H**, `<Zt>H`, `<Pg>/Z`, `<Xn|SP>`, `<Xm>`, LSL #1

```plaintext
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
```

### 32-bit element

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>0</td>
<td>1</td>
<td>0</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>
</tr>
</tbody>
</table>
```

**LD1H**, `<Zt>S`, `<Pg>/Z`, `<Xn|SP>`, `<Xm>`, LSL #1

```plaintext
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
```

### 64-bit element

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>0</td>
<td>1</td>
<td>0</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>
</tr>
</tbody>
</table>
```

**LD1H**, `<Zt>S`, `<Pg>/Z`, `<Xn|SP>`, `<Xm>`, LSL #1

```plaintext
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
```
LD1H \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1] \n
if \!
HaveSVE() then UNDEFINED;  
if \!
Rm == '11111' then UNDEFINED;  
integer t = \!
(UInt)(Zt);  
integer n = \!
(UInt)(Rn);  
integer m = \!
(UInt)(Rm);  
integer g = \!
(UInt)(Pg);  
integer esize = 64;  
integer msize = 16;  
boolean unsigned = TRUE;  

Assembler Symbols  

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.  
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.  
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.  
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.  

Operation  

CheckSVEEnabled();  
integer elements = \!
(VL) DIV esize;  
bits(64) base;  
bits(64) addr;  
bits(PL) mask = \!
P[g];  
bits(VL) result;  
bits(msize) data;  
bits(64) offset = \!
(X[m]);  
constant integer mbytes = msize DIV 8;  

if \!
HaveMTEExt() then \!
SetTagCheckedInstruction(TRUE);  

if n == 31 then  
    if \!
LastActiveElement(mask, esize) >= 0 \|
     Construn.patbleBool(Unpredictable_CHECKSPNONEACTIVE) then \!
     CheckSPAlignment();  
else  
    base = \!
(X[n]);  

for e = 0 to elements-1  
    addr = base + \!
(UInt)(offset) * mbytes;  
    if \!
ElemP[mask, e, esize] == '1' then \!
        data = \!
(Mem[addr, mbytes, AccType_NORMAL];  
        Elem[result, e, esize] = \!
Extend(data, esize, unsigned);  
    else  
        Elem[result, e, esize] = \!
Zeros();  
    offset = offset + 1;  

Z[t] = result;
LD1H (scalar plus vector)

Gather load unsigned halfwords to vector (vector index).

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

|31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 0 0 0 1 0 0 1 | xs | 1 | Zm | 0 | 1 | 0 | Pg | Rn | Zt |

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked scaled offset

|31 30 29 28 27 26 25 24 23 22 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 | 1 | 0 | Pg | Rn | Zt |

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked unscaled offset

|31 30 29 28 27 26 25 24 23 22 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 | 1 | 0 | Pg | Rn | Zt |

msz<1>msz<0> U ff
LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | xs | 0  | Zm | 0  | 1  | 0  | Pg | Rn | Zt |

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | Zm | 1  | 1  | 0  | Pg | Rn | Zt |

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | Zm | 1  | 1  | 0  | Pg | Rn | Zt |

LD1H (scalar plus vector)
LD1H {<Zt>.D}, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
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;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

<table>
<thead>
<tr>
<th>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>1211109876543210</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>dtypeh&lt;1&gt;dtypeh&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td>dtype&lt;1&gt;dtype&lt;0&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>1211109876543210</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>dtypeh&lt;1&gt;dtypeh&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td>dtype&lt;1&gt;dtype&lt;0&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>1211109876543210</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>dtypeh&lt;1&gt;dtypeh&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td>dtype&lt;1&gt;dtype&lt;0&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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(imm6);
64-bit element

<table>
<thead>
<tr>
<th>31302928272625</th>
<th>24</th>
<th>23</th>
<th>22120191817165</th>
<th>14</th>
<th>13</th>
<th>121110</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 0 0 0 0 1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtYPEH<1>dtYPEH<0>  dtYPE<1>dtYPE<0>

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;
bool bool 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, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
inTeger elements = VL DIV esize;
bInts(64) base;
bInts(64) addr;
bInts(PL) mask = P[g];
bInts(VL) result;
bInts(msize) data;
const int mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) == 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
inTeger last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        else
            Elem[result, e, esize] = Zeros();
Z[t] = result;
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.

```
LD1RD {<Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}] 
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 504, defaulting to 0, encoded in the "imm6" field.
Operation

`CheckSVEEnabled();`

integer `elements = VL DIV esize;`

bits(64) `base;`

bits(64) `addr;`

bits(`PL`) `mask = P[g];`

bits(`VL`) `result;`

bits(msize) `data;`

constant integer `mbytes = msize DIV 8;`

if `HaveMTEExt()` then `SetTagCheckedInstruction(TRUE);`

if `n == 31` then
  if `LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = `SP[];`
else
  base = `X[n];`

integer `last = LastActiveElement(mask, esize);`

if `last >= 0` then
  `addr = base + offset * mbytes;`
  `data = Mem[addr, mbytes, AccType_NORMAL];`

for `e = 0` to `elements-1`
  if `ElemP[mask, e, esize] == '1'` then
    `Elem[result, e, esize] = Extend(data, esize, unsigned);`
  else
    `Elem[result, e, esize] = Zeros();`

`Z[t] = result;`
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>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>121110 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0 0 1 1 imm6 1 0 1 Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>121110 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0 0 1 1 imm6 1 1 0 Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>121110 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0 0 1 1 imm6 1 1 1 Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD1RH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #imm}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];
integer last = LastActiveElement(mask, esize);
if last >= 0 then
  addr = base + offset * mbytes;
  data = Mem[addr, mbytes, AccType_NORMAL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1RQB (scalar plus immediate)

Contiguous load and replicate sixteen bytes (immediate index).

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

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

integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSAlignment();
      if HaveMTEEext() then SetTagCheckedInstruction(FALSE);
      base = SP[];
    else
      if HaveMTEEext() then SetTagCheckedInstruction(TRUE);
      base = X[n];
  addr = base + offset * 16;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
LD1RQB (scalar plus immediate)
LD1RQB (scalar plus scalar)

Contiguous load and replicate sixteen bytes (scalar index).

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rm | 0  | 0  | 0  | Pg | Rn | Zt |

msz<1> msz<0> ssz

LD1RQB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    base = SP[];
  else
    base = X[n];
  offset = X[m];
addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
LD1RQD (scalar plus immediate)

Contiguous load and replicate two doublewords (immediate index).

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

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;
b bits(64) base;
b bits(64) addr;
b bits(128) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
  else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * 16;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
LD1RQD (scalar plus immediate)
LD1RQD (scalar plus scalar)

Contiguous load and replicate two doublewords (scalar index).

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>msz&lt;1&gt; msz&lt;0&gt; ssz</td>
</tr>
</tbody>
</table>

LD1RQD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

if ! HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    base = SP[];
  else
    base = X[n];
  offset = X[m];
addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
LD1RQH (scalar plus immediate)

Contiguous load and replicate eight halfwords (immediate index).

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

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

```assembly
CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSParaligment();
      if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
      base = SP[];
    else
      if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
      base = X[n];
  addr = base + offset * 16;
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[result, e, esize] = Zeros();
      addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
```
LD1RQH (scalar plus scalar)

Contiguous load and replicate eight halfwords (scalar index).

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0</td>
<td>1 0 0</td>
<td>Pg Rn Zt ssz msz&lt;1&gt; msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

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 SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    base = SP[];
  else
    base = X[n];
  offset = X[m];
addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
LD1RQW (scalar plus immediate)

Contiguous load and replicate four words (immediate index).

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

```
LD1RQW { <Zt>.S }, <Pg> / Z, [<Xn|SP>{{, #<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 signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

**Operation**

```assembly
CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if n == 31 then  
    if LastActiveElement(mask, esize) >= 0 ||  
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then  
            CheckSPAlignment();  
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);  
            base = SP[];  
        else if HaveMTEExt() then SetTagCheckedInstruction(TRUE);  
            base = X[n];  
    addr = base + offset * 16;  
for e = 0 to elements-1  
    if ElemP[mask, e, esize] == '1' then  
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];  
    else  
        Elem[result, e, esize] = Zeros();  
        addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
```
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 cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

```
LD1RQW { <Zt> }, <Pg>/Z, [<Xn|SP>], <Xm>, LSL #2
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 || ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = X[m];

addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
```
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

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

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

LD1RSB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}

if !HaveSVE() then UNDEFINED;
integer t = UINT(Zt);
integer n = UINT(Rn);
integer g = UINT(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UINT(imm6);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;

bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        base = SP[];
else
    base = X[n];

integer last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1RSH

Load and broadcast signed halfword to vector:

Load a single signed halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | imm6| 1  | 0  | 1  | Pg | Rn | Zt |

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  | 1  | 0  | 1  | imm6| 1  | 0  | 0  | Pg | Rn | Zt |

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

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        base = SP[];
else
    base = X[n];

integer last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
```
LD1RSW

Load and broadcast signed word to vector.

Load a single signed word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

31 0 0 0 1 0 0 1 0 1 1 imm6 1 0 0 0 Pg Rn Zt

dtypeh<1>dtypeh<0> dtypel<1>dtypel<0>

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

`CheckSVEEnabled();`

integer elements = `VL` DIV esize;

bits(64) base;

bits(64) addr;

bits(PL) mask = `P[g];`

bits(VL) result;

bits(msize) data;

constant integer mbytes = msize DIV 8;

if `HaveMTEExt()` then `SetTagCheckedInstruction(TRUE);`

if n == 31 then
  if `LastActiveElement(mask, esize) >= 0 || ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();`
else
  base = `SP[];

integer last = `LastActiveElement(mask, esize);
if last >= 0 then
  addr = base + offset * mbytes;
  data = `Mem[addr, mbytes, AccType_NORMAL];`
for e = 0 to elements-1
  if `ElemP[mask, e, esize] == '1' then `Elem[result, e, esize] = `Extend(data, esize, unsigned);
  else `Elem[result, e, esize] = `Zeros();

Z[t] = result;
**LD1RW**

Load and broadcast unsigned word to vector.

Load a single unsigned word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
[24] 23 2221201918171615 14 13 121110 9 8 7 6 5 4 3 2 1 0
1 0 0 0 1 0 1 0 1 imm6 1 1 0 Pg Rn Zt
```

- `dtypeh<1>`<br>`dtypeh<0>`<br>`dtypel<1>`<br>`dtypel<0>`
- `LD1RW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]`

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

### 64-bit element

```
[24] 23 2221201918171615 14 13 121110 9 8 7 6 5 4 3 2 1 0
1 0 0 0 1 0 1 0 1 imm6 1 1 1 1 Pg Rn Zt
```

- `dtypeh<1>`<br>`dtypeh<0>`<br>`dtypel<1>`<br>`dtypel<0>`
- `LD1RW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]`

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        base = SP[];
else
    base = X[n];

integer last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```
LD1SB (vector plus immediate)

Gather load signed bytes to vector (immediate index).

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

| 31 30 29 28 27 26 25 24 23 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 0 1 1 imm5 1 0 0 0 Pg Zn Zt |

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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|----------------------------------|----------------------------------|
| 1 1 0 0 0 1 0 0 0 1 0 1 1 imm5 1 0 0 0 Pg Zn Zt |

LD1SB \{ <Zt>.D \}, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### Assembler Symbols

- \(<Zt>\) Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- \(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<Zn>\) Is the name of the base scalable vector register, encoded in the "Zn" field.
- \(<imm>\) Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1SB (scalar plus immediate)

Contiguous load signed bytes to vector (immediate index).

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

\[
\text{CheckSVEEnabled}();
\]

integer elements = \( VL \div \text{esize} \);

bits(64) base;

bits(64) addr;

bits(PL) mask = \( P[g] \);

bits(VL) result;

bits(msize) data;

calendar integer mbytes = msize \div 8;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * elements * mbytes;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
        addr = addr + mbytes;

Z[t] = result;
LD1SB (scalar plus scalar)

Contiguous load signed bytes to vector (scalar index).

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element.

16-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|----------------|----------------|----------------|
| 1 0 1 0 0 1 0 1 1 1 0 | 0 | Rm | 0 1 0 | Pg | Rn | Zt |

LD1SB {<Zt>.H}, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;

32-bit element

| 31 30 29 28 27 26 25 24 23 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 | 1 | Rm | 0 1 0 | Pg | Rn | Zt |

LD1SB {<Zt>.S}, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;

64-bit element

| 31 30 29 28 27 26 25 24 23 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 | Rm | 0 1 0 | Pg | Rn | Zt |

LD1SB (scalar plus scalar)
LD1SB { <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 = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;
Z[t] = result;
**LD1SB (scalar plus vector)**

Gather load signed bytes to vector (vector index).

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset.

### 32-bit unpacked unscaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|                |                |                |                |                |                |                |                |
| 0 0 0 0 0 1 0   | 0 0 xs 0       | Zm 0 0 0       | Pg 0           | Rn 0           | Zt 0           | U    ff         |
```

```
LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UINT(Zt);
type n = UINT(Rn);
type m = UINT(Zm);
type g = UINT(Pg);
type size = 64;
type msize = 8;
type of size = 32;
boolean unsigned = FALSE;
boolean of unsigned = xs == '0';
type 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 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|                |                |                |                |                |                |                |                |
| 0 0 0 0 0 1 0   | 0 0 xs 0       | Zm 0 0 0       | Pg 0           | Rn 0           | Zt 0           | U    ff         |
```

```
LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UINT(Zt);
type n = UINT(Rn);
type m = UINT(Zm);
type g = UINT(Pg);
type size = 32;
type msize = 8;
type of size = 32;
boolean unsigned = FALSE;
boolean of unsigned = xs == '0';
type 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 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|                |                |                |                |                |                |                |                |
| 0 0 0 0 0 1 0   | 0 0 1 0        | Zm 1 0 0       | Pg 0           | Rn 0           | Zt 0           | U    ff         |
```

LD1SB (scalar plus vector)
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
    base = SP[];
  else
    base = X[n];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1SH (vector plus immediate)

Gather load signed halfwords to vector (immediate index).

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

```
<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 0 0 0 0 1 0 0</td>
<td>1 0 1</td>
<td>imm5</td>
<td>1 0 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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
<td>U</td>
<td>ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
```

```
LD1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

```java
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

```
<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 1 0 0 0 1 0 0</td>
<td>1 0 1</td>
<td>imm5</td>
<td>1 0 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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
<td>U</td>
<td>ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
```

```
LD1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

```java
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 SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1SH (scalar plus immediate)

Contiguous load signed halfwords to vector (immediate index).

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

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

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

\begin{verbatim}
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
\end{verbatim}
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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1> dtype<0>

LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1> dtype<0>

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == "11111" then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
     ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
    base = SP[];
  else
    base = X[n];
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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 0 0 0 1 0 0 1 xs 1 | Zm | 0 0 0 | Pg | Rn | Zt |

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);
ingter esize = 32;
ingter msize = 16;
ingter offs_size = 32;
ingter unsigned = FALSE;
boolean offs_unsigned = xs == '0';
ingter 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 |

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);
ingter esize = 64;
ingter msize = 16;
ingter offs_size = 32;
ingter unsigned = FALSE;
boolean offs_unsigned = xs == '0';
ingter 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 |

msz<1>msz<0> U ff
LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 32-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | xs | 0  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | Zm | 1  | 0  | 0  | Pg | Rn | Zt |

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;

### 64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | Zm | 1  | 0  | 0  | Pg | Rn | Zt |

LD1SH (scalar plus vector)
LD1SH \{ <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer t = \text{UInt}(Zt);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Zm);
integer g = \text{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.
- \(<\text{mod}>\) Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>(xs)</th>
<th>&lt;\text{mod}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

\text{CheckSVEEnabled}();
integer elements = \text{VL} \text{DIV} esize;
bits(64) base;
bits(64) addr;
bits(\text{VL}) offset = \text{Z}[m];
bits(\text{PL}) mask = \text{P}[g];
bits(\text{VL}) result;
bits(msize) data;
constant integer mbytes = msize \text{DIV} 8;
if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(TRUE);
if n == 31 then
  if \text{LastActiveElement}(mask, esize) >= 0 || \text{ConstrainUnpredictableBool}(\text{Unpredictable_CHECKSPNONEACTIVE}) then
    \text{CheckSPAlignment}();
    base = \text{SP}[];
  else
    base = \text{X}[n];
for e = 0 to elements-1
  if \text{Elem}[mask, e, esize] == '1' then
    integer off = \text{int}(\text{Elem}[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off \ll scale);
    data = \text{Mem}[addr, mbytes, \text{AccType_NORMAL}];
    \text{Elem}[result, e, esize] = \text{Extend}(data, esize, unsigned);
  else
    \text{Elem}[result, e, esize] = \text{Zeros}();
\text{Z}[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1SW (vector plus immediate)

Gather load signed words to vector (immediate index).

Gather load of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

msz<1> msz<0> U ff

LD1SW \{ <Zt>.D }, \langle Pg⟩/Z, \{ <Zn> .D\{, #<imm> \}\}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler Symbols

\(<Zt>\) Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\(<Zn>\) Is the name of the base scalable vector register, encoded in the "Zn" field.
\(<imm>\) Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];
Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
### LD1SW (scalar plus immediate)

Contiguous load signed words to vector (immediate index).

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 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  |

**Asmblr Symbols**

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[n];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements - 1
    if Elem[mask, e, esize] == '1' then
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;
Z[t] = result;
```

LD1SW `{ <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]`
**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 not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1  0  1  0  0  1  0  0  1  0  0  1  0  0  1  0 |    Rn           |     Pg           |     Zt           |

dtype<3:1> dtype<0>
```

LD1SW `{<Zt>.D}, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]`

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
   if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAignment();
   base = SP[];
else
   base = X[n];

for e = 0 to elements-1
   addr = base + UInt(offset) * mbytes;
   if ElemP[mask, e, esize] == '1' then
      data = Mem[addr, mbytes, AccType_NORMAL];
      Elem[result, e, esize] = Extend(data, esize, unsigned);
   else
      Elem[result, e, esize] = Zeros();
   offset = offset + 1;

Z[t] = result;
LD1SW (scalar plus vector)

Gather load signed words to vector (vector index).

Gather load of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit unpacked scaled offset

```
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

```
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

```
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;
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | Zm | 1  | 0  | 0  | Pg | Rn | Zt |

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[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 cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

<table>
<thead>
<tr>
<th>31</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>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>
<td></td>
</tr>
</tbody>
</table>

LD1W { <Zt>.S }, <Pg>/Z, [ <Zn>.S{, #<imm>} ]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### 64-bit element

<table>
<thead>
<tr>
<th>31</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>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>
<td></td>
</tr>
</tbody>
</table>

LD1W { <Zt>.D }, <Pg>/Z, [ <Zn>.D{, #<imm>} ]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
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 not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 0 1 0 0 1 0 0 1 0 1 1 0 0 0 | imm4 | 1 0 1 | Pg | Rn | Zt |

```plaintext
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

| 31 30 29 28 27 26 25 24 23 22 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 1 0 0 0 | imm4 | 1 0 1 | Pg | Rn | Zt |

```plaintext
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
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
end if

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();
  end if
  addr = addr + mbytes;
end for

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 not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '111111' 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>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '111111' 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

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + Uint(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
  Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;

Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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 cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

32-bit unpacked scaled offset

32-bit unpacked unscaled offset
LDIW \{ <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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 0 xs 0 Zm 0 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDIW \{ <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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 0 1 1 Zm 1 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDIW \{ <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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 0 1 1 Zm 1 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDIW (scalar plus vector)
LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  if 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 cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 1 0 1 0 imm4 1 1 1 Pg Rn Zt
```

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

\texttt{CheckSVEEnabled();}
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(\texttt{VL}) values;

if n == 31 then
  if \texttt{LastActiveElement}(mask, esize) >= 0 ||
    \texttt{ConstrainUnpredictableBool}(\texttt{Unpredictable}_CHECKSPNONEACTIVE) then
    \texttt{CheckSPAlignment();}
  if \texttt{HaveMTEExt}() then \texttt{SetTagCheckedInstruction}(FALSE);
  base = \texttt{SP}[];
else
  if \texttt{HaveMTEExt}() then \texttt{SetTagCheckedInstruction}(TRUE);
  base = \texttt{X}[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if \texttt{ElemP}[mask, e, esize] == '1' then
      \texttt{Elem}[values[r], e, esize] = \texttt{Mem}[addr, mbytes, 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];
LD2B (scalar plus scalar)

Contiguous load two-byte structures to two vectors (scalar index).

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

31 30 29 28 27 26 25 24 23 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 1 1 0 1 1 0 0 0 0 0 0 1 1 0 0
msz<1>msz<0>

LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD2D (scalar plus immediate)

Contiguous load two-doubleword structures to two vectors (immediate index).

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & \text{imm4} & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

\(\text{msz<1>msz<0>}\)

LD2D \{ \text{<Zt1>.D}, \text{<Zt2>.D} \}, \text{<Pg>/Z}, [\text{<Xn|SP>\{}{, #\text{<imm>}, \text{MUL VL}}]\}

if \! \text{HaveSVE()} \text{ then UNDEFINED;}
integer t = \text{UInt}(\text{Zt});
integer n = \text{UInt}(\text{Rn});
integer g = \text{UInt}(\text{Pg});
integer esize = 64;
integer offset = \text{SInt}(\text{imm4});
integer 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.
\(<\text{imm}>\) Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

\[ \text{CheckSVEEnabled}(); \]
integer elements = \( \text{VL} \) DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(\text{VL}) values;

if n == 31 then
  if \text{LastActiveElement}(mask, esize) >= 0 || \text{ConstrainUnpredictableBool}('Unpredictable_CHECKSPNONEACTIVE') then
    \text{CheckSPAlignment}();
  if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(FALSE);
  base = \text{SP}[];
else
  if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(TRUE);
  base = \text{X}[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if \text{ElemP}[\text{mask}, e, \text{esize}] == '1' then
      \text{Elem}[values[r], e, \text{esize}] = \text{Mem}[addr, mbytes, AccType_NORMAL];
    else
      \text{Elem}[values[r], e, \text{esize}] = \text{Zeros}();
    addr = addr + mbytes;

for r = 0 to nreg-1
  \text{Z}[(t+r) \mod 32] = values[r];
LD2D (scalar plus scalar)

Contiguous load two-doubleword structures to two vectors (scalar index).

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

LD2D {<Zt1>.D, <Zt2>.D, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]}

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer nreg = 2;

Assembler Symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD2H (scalar plus immediate)

Contiguous load two-halfword structures to two vectors (immediate index).

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

```plaintext
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 = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD2H (scalar plus scalar)

Contiguous load two-halfword structures to two vectors (scalar index).

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

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.

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;
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD2W (scalar plus immediate)

Contiguous load two-word structures to two vectors (immediate index).

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD2W (scalar plus scalar)**

Contiguous load two-word structures to two vectors (scalar index).

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) == 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
        else
            Elem[values[r], e, esize] = Zeros();
        addr = addr + mbytes;
        offset = offset + nreg;
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD3B (scalar plus immediate)

Contiguous load three-byte structures to three vectors (immediate index).

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```
LD3B \{ \langle Zt1\rangle.B, \langle Zt2\rangle.B, \langle Zt3\rangle.B \}, \langle Pg\>/Z, [\langle Xn|SP\rangle\{, \langle 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 = 3;

Assembler Symbols

\(<\text{Zt}1>\) Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
\(<\text{Zt}2>\) Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
\(<\text{Zt}3>\) Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 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.
\(<\text{imm}>\) Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \div \texttt{esize};
bits(64) base;
bits(64) addr;
bits(PL) mask = \texttt{P}[g];
constant integer mbytes = \texttt{esize} \div \texttt{8};
array [0..2] of bits(\texttt{VL}) values;

if n == 31 then
  if LastActiveElement(mask, esize) \geq 0 ||
      ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(\texttt{FALSE});
  base = \texttt{SP}[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(\texttt{TRUE});
  base = \texttt{X}[n];

addr = base + offset \times elements \times nreg \times 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) \mod 32] = values[r];
LD3B (scalar plus scalar)

Contiguous load three-byte structures to three vectors (scalar index).

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```
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 SetTagCheckedInstruction(TRUE);
if n == 31 then
   if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   base = SP[];
else
   base = X[n];
for e = 0 to elements-1
   addr = base + UInt(offset) * mbytes;
   for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1' then
         Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
      else
         Elem[values[r], e, esize] = Zeros();
         addr = addr + mbytes;
         offset = offset + nreg;
for r = 0 to nreg-1
   Z[(t+r) MOD 32] = values[r];
LD3D (scalar plus immediate)

Contiguous load three-doubleword structures to three vectors (immediate index).

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

| 1 0 1 0 0 1 0 | 1 | 1 0 0 | im4 | 1 1 | Pg | Rn | Zt |

LD3D { <Zt1>.D, <Zt2>.D, <Zt3>.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 = 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(VL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPA(Qalignment());
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
  for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD3D (scalar plus scalar)**

Contiguous load three-doubleword structures to three vectors (scalar index).

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

![ASM Symbols]( ASM Symbols.png)

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 SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
  offset = offset + nreg;
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

![Binary representation](image)

```
```

If `!HaveSVE()` then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 3;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL\ DIV\ esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize\ \ DIV\ \ 8;
array [0..2] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 \||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset \times elements \times nreg \times mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
        else
            Elem[values[r], e, esize] = Zeros();
        addr = addr + mbytes;
    for r = 0 to nreg-1
        Z[(t+r) \mod 32] = values[r];
LD3H (scalar plus scalar)

Contiguous load three-halfword structures to three vectors (scalar index).

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

### 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.

```plaintext
LD3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 3;

```plaintext
LD3H (scalar plus scalar)
```
Operation

`CheckSVEEnabled();`
integer elements = \( VL \) DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = \( P[g] \);
bits(64) offset = \( X[m] \);
constant integer mbytes = esize DIV 8;
array [0..2] of bits(\( VL \)) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        base = \( SP[] \);
else
    base = \( X[n] \);

for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
        Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;
for r = 0 to nreg-1
    \( Z[(t+r) MOD 32] = values[r] \);
LD3W (scalar plus immediate)

Contiguous load three-word structures to three vectors (immediate index).

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the “imm4” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
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 cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```assembly
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

```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 SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[ ];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;
  offset = offset + nreg;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD4B (scalar plus immediate)**

Contiguous load four-byte structures to four vectors (immediate index).

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```
```

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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPA(Alignment);
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem(values[r], e, esize] = Mem(addr, mbytes, AccType_NORMAL);
    else
      Elem(values[r], e, esize] = Zeros();
    addr = addr + mbytes;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD4B (scalar plus scalar)

Contiguous load four-byte structures to four vectors (scalar index).

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as “Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as “Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as “Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = \texttt{P}[g];
bits(64) offset = \texttt{X}[m];
constant integer mbytes = esize \texttt{DIV} 8;
array [0..3] of bits(\texttt{VL}) values;

if \texttt{HaveMTEExt}() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if \texttt{LastActiveElement}(mask, esize) >= 0 ||
    \texttt{ConstrainUnpredictableBool}(\texttt{Unpredictable\_CHECKSPNONEACTIVE}) then
    \texttt{CheckSPAlignment}();
  else
    base = \texttt{SP}[];
else
  base = \texttt{X}[n];

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{Elem}[values[r], e, esize] = \texttt{Mem}[addr, mbytes, AccType\_NORMAL];
    else
      \texttt{Elem}[values[r], e, esize] = \texttt{Zeros}();
    addr = addr + mbytes;
    offset = offset + nreg;

for r = 0 to nreg-1
  \texttt{Z}[(t+r) \texttt{MOD} 32] = values[r];
LD4D (scalar plus immediate)

Contiguous load four-doubleword structures to four vectors (immediate index).

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```plaintext
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

**CheckSVEEnabled();**
integer elements = \(VL\) DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = \(P[g]\);
constant integer mbytes = esize DIV 8;
array [0..3] of bits(\(VL\)) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 || ConstrainUnpredictableBool(\(Unpredictable\_CHECKSPNONEACTIVE\)) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = \(SP[\]\);
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = \(X[n]\);

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];
LD4D (scalar plus scalar)

Contiguous load four-doubleword structures to four vectors (scalar index).

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Zt1&gt;</td>
<td>Is the name of the first scalable vector register to be transferred, encoded in the &quot;Zt&quot; field.</td>
</tr>
<tr>
<td>&lt;Zt2&gt;</td>
<td>Is the name of the second scalable vector register to be transferred, encoded as &quot;Zt&quot; plus 1 modulo 32.</td>
</tr>
<tr>
<td>&lt;Zt3&gt;</td>
<td>Is the name of the third scalable vector register to be transferred, encoded as &quot;Zt&quot; plus 2 modulo 32.</td>
</tr>
<tr>
<td>&lt;Zt4&gt;</td>
<td>Is the name of the fourth scalable vector register to be transferred, encoded as &quot;Zt&quot; plus 3 modulo 32.</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;Xm&gt;</td>
<td>Is the 64-bit name of the general-purpose offset register, encoded in the &quot;Rm&quot; field.</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(64) base;
bits(64) addr;
bits(PL) mask = \texttt{P}[g];
bits(64) offset = \texttt{X}[m];
constant integer mbytes = esize \texttt{DIV} 8;
array [0..3] of bits(\texttt{VL}) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = \texttt{SP}[];
else
  base = \texttt{X}[n];

for e = 0 to elements-1
  addr = base + \texttt{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) \texttt{MOD} 32] = values[r];
LD4H (scalar plus immediate)

Contiguous load four-halfword structures to four vectors (immediate index).

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector’s in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

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

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPEAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
  for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD4H (scalar plus scalar)**

Contiguous load four-halfword structures to four vectors (scalar index).

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```
```

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 SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;
  for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
LD4W (scalar plus immediate)

Contiguous load four-word structures to four vectors (immediate index).

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector’s in-memory size, irrespective of predication.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------------------|-------------------------------|-------------------------------|
| 1 0 1 0 0 1 0 1 0 1 1 0 | imm4 | 1 1 | 1 | Pg | Rn | Zt |
```


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 4;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = (VE DIV esize);
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbases = esize DIV 8;
array [0..3] of bits(VE) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbases;
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, mbases, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbases;
  
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
**LD4W (scalar plus scalar)**

Contiguous load four-word structures to four vectors (scalar index).

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```assembly
LD4W

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 = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(P) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for 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];
LDFF1B (vector plus immediate)

Gather load first-fault unsigned bytes to vector (immediate index).

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | imm5 | 1  | 1  | 1  | Pg | Zn | Zt |

msz<1>msz<0> U ff

LDFF1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | imm5 | 1  | 1  | 1  | Pg | Zn | Zt |

msz<1>msz<0> U ff

LDFF1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
      ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
      if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
      elseif 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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 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  |

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

LDFF1B \{ <Zt>.S \}, <Pg>/Z, [<Xn|SP>{, <Xm}>]

if \!HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

\[
dtype<3:1>\dtype<0>
\]

LDFF1B \{ <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 = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a 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;
LDFF1B (scalar plus vector)

Gather load first-fault unsigned bytes to vector (vector index).

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset

### 32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 x 0 Zm 0 1 1 Pg Rn Zt</td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 x 0 Zm 0 1 1 Pg Rn Zt</td>
</tr>
<tr>
<td>U ff</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 0 Zm 1 1 1 Pg Rn Zt</td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

LDFF1B (scalar plus vector)
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>

LDFF1B (scalar plus vector)
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \div \texttt{esize};
bits(64) base;
bits(64) addr;
bits(\texttt{VL}) offset;
bits(\texttt{PL}) mask = \texttt{P}[g];
bits(\texttt{VL}) result;
bits(\texttt{VL}) orig = \texttt{Z}[t];
bits(msize) data;
constant integer mbytes = msize \div 8;
boolean first = \texttt{TRUE};
boolean fault = \texttt{FALSE};
boolean faulted = \texttt{FALSE};
boolean unknown = \texttt{FALSE};
if HaveMTEExt() then SetTagCheckedInstruction(\texttt{TRUE});
if \texttt{n} == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = \texttt{SP}[];
else
  base = \texttt{X}[n];
offset = \texttt{Z}[m];
for \texttt{e} = 0 to elements-1
  if Elem\texttt{F}[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, \texttt{AccType\_NORMAL}];
      first = \texttt{FALSE};
    else
      // Mem\texttt{NF}[] will return fault=\texttt{TRUE} if access is not performed for any reason
      (data, fault) = Mem\texttt{NF}[addr, mbytes, \texttt{AccType\_NONFAULT}];
    else
      (data, fault) = (Zeros(msize), \texttt{FALSE});
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
      Elem\texttt{F}[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || Elem\texttt{F}[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);
  \texttt{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 cause a read from Device memory or signal faults, and are set to zero in the destination vector.

```
1 1 0 0 1 0 1 1 0 1 imm5 1 1 1 Pg Zn Zt
```

LDFF1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & ConstrnUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrnUnpredictableBool(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);
  elem = 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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 1 1 1 1 | Rm | 0 1 1 | Pg | Rn | Zt

dtype<3:1>dtype<0>

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm}, LSL #3]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = base + UInt(offset) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
        else
            (data, fault) = (Zeros(msize), FALSE);
// FFR elements set to FALSE following a suppressed access/fault
faulted = faulted || fault;
if faulted then
    ElemFFR[e, esize] = '0';
// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
    if !fault &&
        ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif
        ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
        Elem[result, e, esize] = Zeros();
    else // 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 cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

### 32-bit unpacked scaled offset

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 1 1 xs 1 Zm 0 1 1 Pg Rn Zt
```

**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;
```

### 32-bit unpacked unscaled offset

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 1 1 xs 0 Zm 0 1 1 Pg Rn Zt
```

**LDFF1D** { <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 = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit scaled offset

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 1 1 1 1 Zm 1 1 1 Pg Rn Zt
```

**LDFF1D** (scalar plus vector)
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | Zm | 1  | 1  | 1  | Pg | Rn | Zt |

LDFF1D { <Z>.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 = 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 *HaveMTExtx*() then *SetTagCheckedInstruction*(TRUE);

if n == 31 then

if *LastActiveElement*(mask, esize) >= 0 ||
  *ConstrainUnpredictableBool*(Unpredictable_CHECKSPNONEACTIVE) then
  *CheckSPAlignment*();

  base = *SP*[];

else

  base = *X*[n];

  offset = *Z*[m];

for e = 0 to elements-1

  if *ElemP*[mask, e, esize] == '1' then
    integer off = *Int*(*Elem*[offset, e, esize]<offs_size-1:0>, offs_unsigned);

    addr = base + (off << scale);

    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = *Mem*[addr, mbytes, AccType_NORMAL];

      first = FALSE;

    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];

    else

      (data, fault) = (*Zeros*(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;

    if faulted then

      *ElemFFR*[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || *ElemFFR*[e, esize] == '0';

    if unknown then

      if !fault &&
        *ConstrainUnpredictableBool*(Unpredictable_SVELDNFDATA) then

        *Elem*[result, e, esize] = *Extend*(data, esize, unsigned);

      elseif

        *ConstrainUnpredictableBool*(Unpredictable_SVELDNFZERO) then

          *Elem*[result, e, esize] = *Zeros*();

      else

        // merge

        *Elem*[result, e, esize] = *Elem*[orig, e, esize];

      else

        *Elem*[result, e, esize] = *Extend*(data, esize, unsigned);

    *Z*[t] = result;
**LDFF1H (vector plus immediate)**

Gather load first-fault unsigned halfwords to vector (immediate index).

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 1 0 1</td>
<td>1 0 1 imm5 1 1 1 Pg Zn Zt</td>
<td></td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
<td>U ff</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

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 1 0 1</td>
<td>1 0 1 imm5 1 1 1 Pg Zn Zt</td>
<td></td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
<td>U ff</td>
<td></td>
</tr>
</tbody>
</table>
```

**LDFF1H**  
\{ <Zt>.D \}, \{<Pg>/Z, \{<Zn>.D{, #<imm>}\}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

**Assembler Symbols**

- **<Zt>**  
  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

- **<Pg>**  
  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- **<Zn>**  
  Is the name of the base scalable vector register, encoded in the "Zn" field.

- **<imm>**  
  Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a 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;
LDFF1H (scalar plus scalar)

Contiguous load first-fault unsigned halfwords to vector (scalar index).

Contiguous load with first-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

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>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

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 = 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>

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.
checkSVEenabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
    base = SP[ ];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault &&
      ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif
      ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  offset = offset + 1;
Z[t] = result;
LDFF1H (scalar plus vector)

Gather load first-fault unsigned halfwords to vector (vector index).

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

32-bit scaled offset

<table>
<thead>
<tr>
<th>Index</th>
<th>Zm</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>xs</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>1</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Zm</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>xs</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>1</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Zm</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>1</td>
<td>xs</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>1</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0>
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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| t                | n                | m                |
| Zt               | Rn               | Zm               |
| xs               | 0                | 1                |

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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| t                | n                | m                |
| Zt               | Rn               | Zm               |
| Zm               | Pg               | Rn               |
| Zt               | U                  |
| Zm               | 1                |
| Zt               | 0                |

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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| t                | n                | m                |
| Zt               | Rn               | Zm               |
| Zm               | Pg               | Rn               |
| Zt               | U                  |
| Zm               | 1                |
| Zt               | 0                |

msz<1>msz<0>
LDFF1H \{ \textless Zt\}.D, \textless Pg\}/Z, [\textless Xn|SP\>, \textless Zm\>.D\}

if \! \texttt{HaveSVE()\} then UNDEFINED;
integer t = \texttt{UInt}(Zt);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Zm);
integer g = \texttt{UInt}(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

**Assembler Symbols**

\texttt{<Zt>} Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
\texttt{<Pg>} Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\texttt{<Xn|SP>} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\texttt{<Zm>} Is the name of the offset scalable vector register, encoded in the "Zm" field.
\texttt{<mod>} Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = Z[m];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
      ElemFFR[e, esize] = '0';
  else
    // 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 (vector plus immediate)

Gather load first-fault signed bytes to vector (immediate index).

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 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  |
| msz<1>msz<0> | U ff |

LDFF1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}] 

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### 64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| msz<1>msz<0> | U ff |

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}] 

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### Assembler Symbols

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the base scalable vector register, encoded in the "Zn" field.
- **<imm>** Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

```cpp
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem;base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && Constr unUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif Constr unUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else  // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDFF1SB (scalar plus scalar)

Contiguous load first-fault signed bytes to vector (scalar index).

Contiguous load with first-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

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 size = 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 size = 32;
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 size = 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

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[ ];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  offset = offset + 1;
Z[t] = result;
**LDFF1SB (scalar plus vector)**

Gather load first-fault signed bytes to vector (vector index).

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset.

### 32-bit unpacked unscaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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 | U  | f  |
```

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;

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;

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

### 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  | xs | 0  | Zm | 0  | 0  | 1  | Pg | Rn | Zt | U  | f  |
```

### 64-bit unscaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | Zm | 1  | 0  | 1  | Pg | Rn | Zt | U  | f  |
```

msz<1>msz<0>
LDFF1SB \{ <Zt>.D \}, <Prg>/Z, [<Xn|SP>, <Zm>.D]

if !\texttt{HaveSVE}() then UNDEFINED;
integer t = \texttt{UInt}(Zt);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Zm);
integer g = \texttt{UInt}(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

\texttt{<Zt>}  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
\texttt{<Pg>}  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\texttt{<Xn|SP>}  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\texttt{<Zm>}  Is the name of the offset scalable vector register, encoded in the "Zm" field.
\texttt{<mod>}  Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
LDFF1SH (vector plus immediate)

Gather load first-fault signed halfwords to vector (immediate index).

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
0 1 0 0 0 1 0 | 0 1 0 1 | imm5 1 0 1 | Pg | Zn | Zt
msz<1>msz<0> U ff
```

```java
LDFF1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #imm}]
```

```java
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

```
1 1 0 0 0 1 0 | 0 1 0 1 | imm5 1 0 1 | Pg | Zn | Zt
msz<1>msz<0> U ff
```

```java
LDFF1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #imm}]
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if 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;
**LDFF1SH (scalar plus scalar)**

Contiguous load first-fault signed halfwords to vector (scalar index).

Contiguous load with first-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
31 30 29 28 27 26 25 24 23 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 1 | Rm | 0 1 1 | Pg | Rn | Zt
```

**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

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 1 0 0 0 | Rm | 0 1 1 | Pg | Rn | Zt
```

**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

```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 mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  offset = offset + 1;
Z[t] = result;
```
LDFF1SH (scalar plus vector)

Gather load first-fault signed halfwords to vector (vector index).

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 xs 1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 xs 0</td>
</tr>
</tbody>
</table>

msz<1>msz<0> __U ff
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | xs | 0  | Zm | 0  | 0  | 1  | Pg | Rn | Zt | U | ff |

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | Pg | Rn | Zt | U | ff |

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | Zm | 1  | 0  | 1  |Pg | Rn | Zt | U | ff |

LDFF1SH (scalar plus vector)
if !\texttt{HaveSVE}() then UNDEFINED;
integer t = \texttt{UInt(Zt)};
integer n = \texttt{UInt(Rn)};
integer m = \texttt{UInt(Zm)};
integer g = \texttt{UInt(Pg)};
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

\textbf{Assembler Symbols}

\begin{itemize}
\item \texttt{<Zt>} Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
\item \texttt{<Pg>} Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\item \texttt{<Xn|SP>} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\item \texttt{<Zm>} Is the name of the offset scalable vector register, encoded in the "Zm" field.
\item \texttt{<mod>} Is the index extend and shift specifier, encoded in "xs":
\end{itemize}

\begin{tabular}{|c|c|}
\hline
\texttt{xs} & \texttt{<mod>} \\
\hline
0 & UXTW \\
1 & SXTW \\
\hline
\end{tabular}
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
// FFR elements set to FALSE following a supressed access/fault
faulted = faulted || fault;
if faulted then
  ElemFFR[e, esize] = '0';
// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
  if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
    Elem[result, e, esize] = Zeros();
  else  // merge
    Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t] = result;
```
LDFF1SW (vector plus immediate)

Gather load first-fault signed words to vector (immediate index).

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

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 SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
```
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 not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  1 0 1 0 0 1 0 | 0 1 0 | 0 | Rm | 0 1 1 | Pg | Rn | Zt
```

dtype<3:1>dtype<0>

LDFF1SW { <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 = 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 mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
      elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
        Elem[result, e, esize] = Zeros();
      else
        Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    offset = offset + 1;
  Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDFF1SW (scalar plus vector)

Gather load first-fault signed words to vector (vector index).

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4.Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | xs | 1  | Zm | 0  | 0  | 1  | Pg | Rn | Zt |

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  | 1  | 0  | 1  | 0  | xs | 0  | Zm | 0  | 0  | 1  | Pg | Rn | Zt |

LDF1SW { <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 = 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  | 1  | Pg | Rn | Zt |

LDF1SW (scalar plus vector)
LDFF1SW \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | Zm | 1  | 0  | 1  | Pg | Rn | Zt |

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 "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 SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
offset = Z[m];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault &&
            ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
LDFF1W (vector plus immediate)

Gather load first-fault unsigned words to vector (immediate index).

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 | msz<1>msz<0> | U | ff |

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  | 1  | 0  | 1  | 0  | 0  | 1  | imm5| 1  | 1  | 1  | Pg | Zn | Zt | msz<1>msz<0> | U | ff |

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 = \texttt{VL} \div esize;
bits(\texttt{VL}) base = \texttt{Z}[n];
bits(64) addr;
bits(\texttt{PL}) mask = \texttt{P}[g];
bits(\texttt{VL}) result;
bits(\texttt{VL}) orig = \texttt{Z}[t];
bits(msize) data;
constant integer mbytes = msize \div 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    end
    endif
    (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';
    endif
    // 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);
      elseif 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
  end
\texttt{Z}[t] = result;
LDFF1W (scalar plus scalar)

Contiguous load first-fault unsigned words to vector (scalar index).

Contiguous load with first-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

dtype<3:1>dtype<0>

LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm}, LSL #2}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;

64-bit element

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

dtype<3:1>dtype<0>

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm}, LSL #2}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE); 
if n == 31 then 
   if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   base = SP[];
else
   base = X[n];
for e = 0 to elements-1 
   if ElemP[mask, e, esize] == '1' then
      addr = base + UInt(offset) * mbytes;
      if first then
         // Mem[] will not return if a fault is detected for the first active element
         data = Mem[addr, mbytes, AccType_NORMAL];
         first = FALSE;
      else
         // MemNF[] will return fault=TRUE if access is not performed for any reason
         (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
      else
         (data, fault) = (Zeros(msize), FALSE);
      offset = offset + 1;
   else
      // FFR elements set to FALSE following a suppressed access/fault
      if fault then
         ElemFFR[e, esize] = '0';
      else
         // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
         unknown = unknown || ElemFFR[e, esize] == '0';
      if unknown then
         if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
         elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
         else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
      else
         Elem[result, e, esize] = Extend(data, esize, unsigned);
      offset = offset + 1;
   Z[t] = result;
LDFF1W (scalar plus vector)

Gather load first-fault unsigned words to vector (vector index).

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

### 32-bit scaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 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               | Zm             | 0 1 1           | Pg             |
| Zt                             |                |                |                |
```

LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

```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 = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
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 1 0               | Zm             | 0 1 1           | Pg             |
| Zt                             |                |                |                |
```

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

```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 = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
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 1 0 1 0               | Zm             | 0 1 1           | Pg             |
| Zt                             |                |                |                |
```

msz<1>msz<0>
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

\[
\begin{array}{cccccccccccccccccccccccccccc}
\end{array}
\]

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

\[
\begin{array}{cccccccccccccccccccccccccccc}
\end{array}
\]

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

\[
\begin{array}{cccccccccccccccccccccccccccc}
\end{array}
\]

LDFF1W (scalar plus vector)
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 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 SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[1];
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);
    else
        // 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 if
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        end if
Z[t] = result;
LDNF1B

Contiguous load non-fault unsigned bytes to vector (immediate index).

Contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

\[
\begin{array}{cccccccccccccccccccccccccccc}
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & \text{imm4} & 1 & 0 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

dtype<3:1>dtype<0>

LDNF1B \{ <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

16-bit element

\[
\begin{array}{cccccccccccccccccccccccccccc}
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \text{imm4} & 1 & 0 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

dtype<3:1>dtype<0>

LDNF1B \{ <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

\[
\begin{array}{cccccccccccccccccccccccccccc}
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \text{imm4} & 1 & 0 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

dtype<3:1>dtype<0>

LDNF1B \{ <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
LDNF1B \{ <Zt> \cdot 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];
bv8 bits(`VL`) result;
bv8 bits(`VL`) orig = `Z`[t];
bv8 bits(`msize`) data;
constant integer mbytes = `msize` DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if `n` == 31 then
  if `ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)` then
    `CheckSPAlignment();`
  if `HaveMTEExt()` then `SetTagCheckedInstruction(FALSE);`
  base = `SP`[];
else
  if `HaveMTEExt()` then `SetTagCheckedInstruction(TRUE);`
  base = `X`[n];

addr = base + offset * elements * `mbytes`;
for `e` = 0 to `elements`-1
  if `ElemP[mask, e, esize] == '1'` then
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, `AccType_NONFAULT`];
  else
    (data, fault) = (`Zeros`(`msize`), FALSE);

  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    `ElemFFR[e, esize] = '0';`

  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || `ElemFFR[e, esize] == '0';`
  if unknown then
    if fault & `ConstrainUnpredictableBool(Constrain_SVELDNFDATA)` then
      `Elem[result, e, esize] = Extend(data, esize, unsigned);`
    elsif `ConstrainUnpredictableBool(Constrain_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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
| 1 0 1 0 0 0|1|1|1|1|imm4|1 0 1| Pg | Rn | Zt |
   dtype<3:1> dtype<0>
```

The assembler code is as follows:

```assembly
LDNF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
  if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAInline();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif 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 not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 0 1 0 | 1 | imm4 | 1 0 1 | Pg | Rn | Zt

dtype<3:1> dtype<0>

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

31 30 29 28 27 26 25 24 23 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 | 1 | imm4 | 1 0 1 | Pg | Rn | Zt

dtype<3:1> dtype<0>

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

31 30 29 28 27 26 25 24 23 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 | imm4 | 1 0 1 | Pg | Rn | Zt

dtype<3:1> dtype<0>

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;
bids(64) base;
bids(64) addr;
bids(PL) mask = P[g];
bids(VL) result;
bids(VL) orig = Z[t];
bids(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
    if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then
           SetTagCheckedInstruction(FALSE);
    else
        if HaveMTEExt() then
           SetTagCheckedInstruction(TRUE);
    base = X[n];
else
    if HaveMTEExt() then
           SetTagCheckedInstruction(FALSE);
    base = SP[];

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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNF1SB

Contiguous load non-fault signed bytes to vector (immediate index).

Contiguous load with non-faulting behavior of signed bytes to elements of a vector register from the memory address
generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read
from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: **16-bit element**, **32-bit element** and **64-bit element**

### 16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21</th>
<th>20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 1</td>
<td>0 1</td>
<td>imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDNF1SB {<Zt>.H }, <Pg>/Z, [{<Xn|SP}>{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21</th>
<th>20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 0 1 1</td>
<td>1 1</td>
<td>imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDNF1SB {<Zt>.S }, <Pg>/Z, [{<Xn|SP}>{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

### 64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21</th>
<th>20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 0 0</td>
<td>1 1</td>
<td>imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDNF1SB {<Zt>.D }, <Pg>/Z, [{<Xn|SP}>{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
  if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    // 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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>imm4</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
<td></td>
<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>32-bit element</td>
<td></td>
<td></td>
<td></td>
<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>

LDNF1SH { <Zt>.S }, <Pg>/Z, [{<Xn|SP>}, #<imm>, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>imm4</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
<td></td>
<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>64-bit element</td>
<td></td>
<td></td>
<td></td>
<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>

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

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if n == 31 then
    if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a 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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 1 0 1 0 0 1 0 | 0 1 0 1 | imm4 | 1 0 1 | Pg | Rn | Zt |

dtype<3:1>dtype<0>

LDNF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if n == 31 then
    if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        // MemNF[] will return fault=True if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
if faulted then
    ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif 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;
**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 not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

**32-bit element**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

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  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

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 = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
  if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  addr = addr + mbytes;
Z[t] = result;
LDNT1B (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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

---

**Assembler Symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
else
    addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;
Z[t] = result;
```

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
**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 not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

![Assembly Code](image)

**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) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) == 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSNONEACTIVE) 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 (scalar plus immediate)

Contiguous load non-temporal doublewords to vector (immediate index).

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

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
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = result;
LDNT1D (scalar plus scalar)

Contiguous load non-temporal doublewords to vector (scalar index).

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 1 0 0 1 0 1 1 0 0 | Rm | 1 1 0 | Pg | Rn | Zt |

msz<1>msz<0>

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(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    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 (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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

![Binary representation of LDNT1H](image)

```assembly
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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[0];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
    endif
else
    addr = base + offset * elements * mbytes;
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
        else
            Elem[result, e, esize] = Zeros();
        endif
        addr = addr + mbytes;
    endfor
    Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
LDNT1H (scalar plus scalar)

Contiguous load non-temporal halfwords to vector (scalar index).

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

```
LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Assembler Symbols

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
offset = X[m];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
    else
        Elem[result, e, esize] = Zeros();
    offset = offset + 1;
Z[t] = result;
```
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 not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

```
            31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
            1  0  1  0  0  1  0  1  0  0  0  0  0  0  imm4  1  1  1  Pg  Rn  Zt
            msz<1>msz<0>

LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
else
    addr = base + offset * elements * mbytes;
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
        else
            Elem[result, e, esize] = Zeros();
        addr = addr + mbytes;
    Z[t] = result;
```
**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 cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0>

LDNT1W \{ <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '1111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 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(VL) mask = P[g];
bits(VL) result;
countant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
basis = SP[];
else
  base = X[n];
  offset = X[m];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;

Z[t] = result;
LDR (predicate)

Load predicate register.

Load a predicate register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated. The load is performed as a stream of bytes containing 8 consecutive predicate bits in ascending element order, without any endian conversion.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | imm9h| 0  | 0  | 0  | imm9l| Rn | 0  | Pt |

LDR <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler Symbols

<Pt> Is the name of the destination scalable predicate register, encoded in the "Pt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
integer elements = PL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(PL) result;
if n == 31 then
    CheckSPALignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_NORMAL, FALSE);
for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_NORMAL, aligned];
    offset = offset + 1;

P[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDR (vector)

Load vector register.

Load a vector register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The load is performed as a stream of byte elements in ascending element order, without any endian conversion.

LDR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

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 HaveMTEEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_NORMAL, FALSE);
for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_NORMAL, aligned];
    offset = offset + 1;

Z[t] = result;
**LSL (immediate, predicated)**

Logical shift left by immediate (predicated).

Shift left by immediate each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

```plaintext
LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

If `HaveSVE()` then UNDEFINED;

```plaintext
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
  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**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

The `MOVPRFX` instruction must specify the same destination register as this instruction.

The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSL (wide elements, predicated)

Logical shift left by 64-bit wide elements (predicated).

Shift left active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

```
0 0 0 0 0 1 0 0 size 0 1 1 0 1 1 1 0 0 Pg Zm Zdn
```

LSL <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>.D

```
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><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>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();
to 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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

LSL (wide elements, predicated)
**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 | 0 1 0 0 1 1 0 0 | Pg | Zm | Zdn |

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

if !HaveSVE() then UNDEFINED;

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] = LSL(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

| 31 30 29 28 27 26 25 24 23 22 21 20 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 | tszl | imm3 | 1 0 0 1 1 | Zn | Zd |

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>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 0 to number of bits per element minus 1, encoded in "tsz:imm3".

**Operation**

```plaintext
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;
```

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;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | size | 1 | Zm | 1 0 0 0 1 1 | Zn | Zd |

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;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LSLR

Reversed logical shift left by vector (predicated).

Reversed shift left active elements of the second source vector by corresponding elements of the first source vector
and destructively place the results in the corresponding elements of the first source vector. The shift amount operand
is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive
elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0        | size            | Pg              |
|                         | 0 1 0 1 1 1 0 0 | Zm              |
|                         |                 | Zdn             |

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 Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSL(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**LSR (immediate, predicated)**

Logical shift right by immediate (predicated).

Shift right by immediate, inserting zeroes, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

![Shift Register](image)

**Assembler Symbols**

- `<Zdn>` is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in “tszh:tszl”:
  
<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<const>` is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**LSR (wide elements, predicated)**

Logical shift right by 64-bit wide elements (predicated).

Shift right, inserting zeroes, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

![Register Bit Positions](image)

**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><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>00</code></td>
</tr>
<tr>
<td><code>01</code></td>
</tr>
<tr>
<td><code>10</code></td>
</tr>
<tr>
<td><code>11</code></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(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
  integer shift = Min(UInt(element2), esize);
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSR (vectors)

Logical shift right by vector (predicated).

Shift right, inserting zeroes, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

### 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

```assembly
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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSR (immediate, unpredicated)

Logical shift right by immediate (unpredicated).

Shift right by immediate, inserting zeroes, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

<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>tszh</td>
<td>1</td>
<td>tszl</td>
<td>imm3</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>
</tr>
</tbody>
</table>

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);

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 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;
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.

```
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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & Pg & Zm & Zdn \\
\end{array}
\]

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 Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSR(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Multiply-add vectors (predicated), writing multiplicand \[ Z_{dn} = Z_a + Z_{dn} \times Z_m \].

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | Zm | 1  | 1  | 0  | Pg | Za | Zdn |

MAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = 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(VL) 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 Elem[mask, e, esize] == '1' then
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Multiply-add vectors (predicated), writing addend \([Zda = Zda + Zn \times Zm]\).

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.

\[
\begin{array}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 1 & 0 & \text{Pg} & \text{Zn} & \text{Zda} & \text{op}
\end{array}
\]

MLA \(<\text{Zda}>,.<\text{T}>, \text{Pg}>/M, <\text{Zn}>,.<\text{T}>, <\text{Zm}>,.<\text{T}>

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 <<< \text{UInt}(\text{size});
integer g = \text{UInt}(\text{Pg});
integer n = \text{UInt}(\text{Zn});
integer m = \text{UInt}(\text{Zm});
integer da = \text{UInt}(\text{Zda});
boolean sub_op = FALSE;

**Assemble Symbols**

<\text{Zda}> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<\text{T}> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\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{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} \text{DIV esize};
bits(\text{PL}) mask = \text{P}[g];
bits(\text{VL}) operand1 = \text{Z}[n];
bits(\text{VL}) operand2 = \text{Z}[m];
bits(\text{VL}) operand3 = \text{Z}[da];
bits(\text{VL}) result;

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{Elem}[\text{mask}, e, \text{esize}] == '1' then
    integer product = element1 \times element2;
    if sub_op then
        \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}[\text{da}] = \text{result};
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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.

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 | size | 0 | Zm | 0 1 1 | Pg | Zn | Zda
```

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**

```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
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV (predicate, predicated, zeroing)

Move predicates (zeroing).

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of AND, ANDS (predicates). This means:

- The encodings in this description are named to match the encodings of AND, ANDS (predicates).
- The description of AND, ANDS (predicates) gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| S | 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 | Pm | 0 1 | Pg | 0 | Pn | 0 | Pd |

MOV <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when $S == '0' \&\& Pn == Pm.$

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of AND, ANDS (predicates) gives the operational pseudocode for this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**MOV (immediate, predicated, zeroing)**

Move signed integer immediate to vector elements (zeroing).

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of **CPY (immediate, zeroing)**. This means:

- The encodings in this description are named to match the encodings of **CPY (immediate, zeroing)**.
- The description of **CPY (immediate, zeroing)** gives the operational pseudocode for this instruction.

```plaintext
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M
```

MOV <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

is equivalent to

CPY <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

and is always the preferred disassembly.

**Assembler Symbols**

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in "size":

  ```plaintext
  size | <T> |
  ----|-----|
  00  | B   |
  01  | H   |
  10  | S   |
  11  | D   |
  ``

- `<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":

  ```plaintext
  sh | <shift> |
  ---|---------|
  0  | LSL #0  |
  1  | LSL #8  |
  ``

**Operation**

The description of **CPY (immediate, zeroing)** gives the operational pseudocode for this instruction.
MOV (immediate, predicated, merging)

Move signed integer immediate to vector elements (merging).

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is \( \# <\text{imm8}> , \text{LSL} \ #8 \). 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 \( \#0 , \text{LSL} \ #8 \).

This is an alias of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccccccccccccc}
\end{array}
\]

<table>
<thead>
<tr>
<th>MOV</th>
<th>(&lt;Zd&gt;.&lt;T&gt;), (&lt;Pg&gt;/M, #&lt;imm&gt;{, &lt;shift}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>is equivalent to CPY (&lt;Zd&gt;.&lt;T&gt;), (&lt;Pg&gt;/M, #&lt;imm&gt;{, &lt;shift}&gt;)</td>
</tr>
<tr>
<td></td>
<td>and is always the preferred disassembly.</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

\(<Zd>\)  
Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\)  
Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Pg>\)  
Is the name of the governing scalable predicate register, encoded in the "Pg" field.

\(<imm>\)  
Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

\(<shift>\)  
Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>(&lt;shift&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV (scalar, predicated)

Move general-purpose register to vector elements (predicated).

Move the general-purpose scalar source register to each active element in the destination vector. Inactive elements in
the destination vector register remain unmodified.

This is an alias of CPY (scalar). This means:

• The encodings in this description are named to match the encodings of CPY (scalar).
• The description of CPY (scalar) gives the operational pseudocode for this instruction.

```
|   31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  |

MOV <Zd>.<T>, <Pg>/M, <R><n|SP>
```

is equivalent to

```
CPY <Zd>.<T>, <Pg>/M, <R><n|SP>
```

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>H</td>
<td>S</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>01</th>
<th>x0</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>W</td>
<td>X</td>
</tr>
</tbody>
</table>

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

The description of CPY (scalar) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
register of this instruction.
MOV (SIMD&FP scalar, predicated)

Move SIMD&FP scalar register to vector elements (predicated).

Move the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of CPY (SIMD&FP scalar). This means:

- The encodings in this description are named to match the encodings of CPY (SIMD&FP scalar).
- The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

\[
\text{MOV } <Zd>, <T>, <Pg>/M, <V><n>
\]

is equivalent to

\[
\text{CPY } <Zd>, <T>, <Pg>/M, <V><n>
\]

and is always the preferred disassembly.

Assembler Symbols

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “size”:
  \[
  \begin{array}{c|c}
  \text{size} & \text{<T>} \\
  \hline
  00 & B \\
  01 & H \\
  10 & S \\
  11 & D \\
  \end{array}
  \]
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<V>** Is a width specifier, encoded in “size”:
  \[
  \begin{array}{c|c}
  \text{size} & \text{<V>} \\
  \hline
  00 & B \\
  01 & H \\
  10 & S \\
  11 & D \\
  \end{array}
  \]
- **<n>** Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

Operation

The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV (immediate, unpredicated)

Move signed immediate to vector elements (unpredicated).

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The description of DUP (immediate) gives the operational pseudocode for this instruction.

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|}
\hline
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & \text{size} & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & \text{sh} & \text{imm8} & \text{Zd} \\
\hline
\end{array}
\]

MOV <Zd>.<T>, #<imm>{, <shift>}

is equivalent to

DUP <Zd>.<T>, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.
**MOV (scalar, unpredicated)**

Move general-purpose register to vector elements (unpredicated).

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This is an alias of **DUP (scalar)**. This means:

- The encodings in this description are named to match the encodings of **DUP (scalar)**.
- The description of **DUP (scalar)** gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 1 0 1 | size | 1 0 0 0 0 0 0 0 1 1 1 0 | Rn | Zd |

**MOV <Zd>,<T>,<R>|n|SP>**

is equivalent to

**DUP <Zd>,<T>,<R>|n|SP>**

and is always the preferred disassembly.

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<R>** Is a width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

- **<n|SP>** Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

**Operation**

The description of **DUP (scalar)** gives the operational pseudocode for this instruction.
MOV (SIMD&FP scalar, unpredicated)

Move indexed element or SIMD&FP scalar to vector (unpredicated).

Unconditionally broadcast the SIMD&FP scalar into each element of the destination vector. This instruction is unpredicated.

This is an alias of DUP (indexed). This means:

- The encodings in this description are named to match the encodings of DUP (indexed).
- The description of DUP (indexed) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & \text{imm2} & 1 & \text{tsz} & 0 & 0 & 1 & 0 & 0 & 0 & \text{Zn} & \text{Zd} & \text{Zd} \\
\end{array}
\]

MOV <Zd>.<T>, <V><n>

is equivalent to

DUP <Zd>.<T>, <Zn>.<T>[0]

and is the preferred disassembly when BitCount(imm2:tsz) == 1.

MOV <Zd>.<T>, <Zn>.<T>[<imm>]

is equivalent to

DUP <Zd>.<T>, <Zn>.<T>[<imm>]

and is the preferred disassembly when BitCount(imm2:tsz) > 1.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsz":

\[
\begin{array}{c|c}
\text{tsz} & \text{T} \\
00000 & \text{RESERVED} \\
xxxx1 & B \\
xxx10 & H \\
x1000 & S \\
x1000 & D \\
10000 & Q \\
\end{array}
\]

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".

<V> Is a width specifier, encoded in "tsz":

\[
\begin{array}{c|c}
\text{tsz} & \text{V} \\
00000 & \text{RESERVED} \\
xxxx1 & B \\
xxx10 & H \\
x1000 & S \\
x1000 & D \\
10000 & Q \\
\end{array}
\]

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Zn" field.

Operation

The description of DUP (indexed) gives the operational pseudocode for this instruction.
MOV (bitmask immediate)

Move logical bitmask immediate to vector (unpredicated).

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This is an alias of DUPM. This means:

- The encodings in this description are named to match the encodings of DUPM.
- The description of DUPM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>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>1110x</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>

MOV <Zd>.<T>, #<const>

is equivalent to

DUPM <Zd>.<T>, #<const>

and is the preferred disassembly when SVEMoveMaskPreferred(imm13).

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Operation

The description of DUPM gives the operational pseudocode for this instruction.
MOV (predicate, unpredicated)

Move predicate (unpredicated).

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Does not set the condition flags.

This is an alias of ORR, ORRS (predicates). This means:

- The encodings in this description are named to match the encodings of ORR, ORRS (predicates).
- The description of ORR, ORRS (predicates) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & Pm & 0 & 1 & Pg & 0 & Pn & 0 & 0 & Pd \\
\end{array}
\]

MOV <Pd>.B, <Pn>.B

is equivalent to


and is the preferred disassembly when \( S == '0' && Pn == Pm && Pm == Pg \).

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of ORR, ORRS (predicates) gives the operational pseudocode for this instruction.
MOV (vector, unpredicated)

Move vector register (unpredicated).

Move vector register. This instruction is unpredicated.

This is an alias of **ORR (vectors, unpredicated)**. This means:

- The encodings in this description are named to match the encodings of **ORR (vectors, unpredicated)**.
- The description of **ORR (vectors, unpredicated)** gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & \text{Zm} & 0 & 0 & 1 & 1 & 0 & 0 & \text{Zn} & \text{Zd}
\end{array}
\]

MOV `<Zd>`.D, `<Zn>`.D

is equivalent to

ORR `<Zd>`.D, `<Zn>`.D, `<Zn>`.D

and is the preferred disassembly when `Zn == Zm`.

**Assembler Symbols**

`<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.

`<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.

**Operation**

The description of **ORR (vectors, unpredicated)** gives the operational pseudocode for this instruction.
MOV (predicate, predicated, merging)

Move predicates (merging).

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register remain unmodified. Does not set the condition flags.

This is an alias of SEL (predicates). This means:

- The encodings in this description are named to match the encodings of SEL (predicates).
- The description of SEL (predicates) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccc}
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & Pm & 0 & 1 & Pg & 1 & Pn & 1 & 1 & \\
\end{array}
\]

MOV <Pd>.B, <Pg>/M, <Pn>.B

is equivalent to


and is the preferred disassembly when \( Pd == Pm \).

Assembler Symbols

\(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.

\(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of SEL (predicates) gives the operational pseudocode for this instruction.
MOV (vector, predicated)

Move vector elements (predicated).

Move elements from the source vector to the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of SEL (vectors). This means:

- The encodings in this description are named to match the encodings of SEL (vectors).
- The description of SEL (vectors) gives the operational pseudocode for this instruction.

Assembler Symbols

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

The description of SEL (vectors) gives the operational pseudocode for this instruction.
Move prefix (predicated).

The predicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically. Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that for performance reasons software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

Although the operation of the instruction is defined as a simple predicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same predicate register, and have the same maximum element size (ignoring a fixed 64-bit "wide vector" operand), and the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

```
MOVPRFX <Zd>.<T>, <Pg>/<ZM>, <Zn>.<T>
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean merging = (M == '1');
```

Assembler Symbols

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<ZM>` is the predication qualifier, encoded in "M":

<table>
<thead>
<tr>
<th>M</th>
<th>&lt;ZM&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the source scalable vector register, encoded in the "Zn" field.
Operation

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();
endfor

Z[d] = result;
MOVRFX (unpredicated)

Move prefix (unpredicated).

The unpredicated MOVRFX 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 MOVRFX instructions. The prefixed instruction must specify the same destination vector as the MOVRFX 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.

MOVRFX <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;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**MOV$S (predicated)**

Move predicates (zeroing), setting the condition flags.

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of **AND, ANDS (predicates)**. This means:

- The encodings in this description are named to match the encodings of **AND, ANDS (predicates)**.
- The description of **AND, ANDS (predicates)** gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
Pm & 0 & 1 & Pg & 0 & Pn & 0 & Pd & 1 & 0 \\
\end{array}
\]

MOV$S <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when \$ == '1' && Pn == Pm.

**Assembler Symbols**

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Pg>** Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- **<Pn>** Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of **AND, ANDS (predicates)** gives the operational pseudocode for this instruction.
**MOVS (unpredicated)**

Move predicate (unpredicated), setting the condition flags.

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Sets the **FIRST** (N), **NONE** (Z), **!LAST** (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of **ORR, ORRS (predicates)**. This means:

- The encodings in this description are named to match the encodings of **ORR, ORRS (predicates)**.
- The description of **ORR, ORRS (predicates)** gives the operational pseudocode for this instruction.

![Predicate Register Encoding](image)

MOVS $<Pd>.B$, $<Pn>.B$

is equivalent to


and is the preferred disassembly when $S == '1' \land Pn == Pm \land Pm == Pg$.

**Assembler Symbols**

- $<Pd>$ Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- $<Pn>$ Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of **ORR, ORRS (predicates)** gives the operational pseudocode for this instruction.
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.

```
0 0 0 0 0 1 0 0  size 0  Zm  1 1 1  Pg  Za  Zdn  op
```

**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**

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P) mask = P[g];
bv_true(on) operand1 = Z[dn];
bv_true(on) operand2 = Z[m];
bv_true(on) operand3 = Z[a];
bv_true(on) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if !ElementP[mask, e, esize] 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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MUL (vectors)

Multiply vectors (predicated).

Multiply active elements of the first source vector by corresponding elements of the second source vector and
destructively place the results in the corresponding elements of the first source vector. Inactive elements in the
destination vector register remain unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 0  | 0  | 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  |
```

H U

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MUL (immediate)

Multiply by immediate (unpredicated).

Multiply by an immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

```
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": |
| size  | <T>  |
| 00    | B    |
| 01    | H    |
| 10    | S    |
| 11    | D    |
| <imm> | Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field. |

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bis(VL) operand1 = Z[dn];
bis(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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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

\[
\begin{array}{cccccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & Pm & 0 & 1 & Pg & 1 & Pn & 1 & Pd & S \\
\end{array}
\]


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

\[
\begin{array}{cccccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & Pm & 0 & 1 & Pg & 1 & Pn & 1 & Pd & S \\
\end{array}
\]


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 AND element2);
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
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.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 0 0 0 1 0 0 size | 0 1 0 1 1 1 0 1 | Pg | Zn | Zd
```

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 1 0 0 1 0 1 1 0 0 1 0 | Pm | 0 1 | Pg | 1 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 1 0 0 1 0 1 1 1 0 0 0 | Pm | 0 1 | Pg | 1 0 | Pn | 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.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PPL) mask = P[g];
bits(PPL) operand1 = P[n];
bits(PPL) operand2 = P[m];
bits(PPL) 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;
NOT (predicate)

Bitwise invert predicate.

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of FOR, EORS (predicates). This means:

- The encodings in this description are named to match the encodings of FOR, EORS (predicates).
- The description of FOR, EORS (predicates) gives the operational pseudocode for this instruction.

\[
\text{NOT} \ <\text{Pd}>.B, <\text{Pg}>/Z, <\text{Pn}>.B
\]

is equivalent to

\[
\text{EOR} \ <\text{Pd}>.B, <\text{Pg}>/Z, <\text{Pn}>.B, <\text{Pg}>.B
\]

and is the preferred disassembly when \( \text{Pm} == \text{Pg} \).

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.

Operation

The description of FOR, EORS (predicates) gives the operational pseudocode for this instruction.
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.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | Pg | Zn | Zd |

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
NOTS

Bitwise invert predicate, setting the condition flags.

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of EOR, EORS (predicates). This means:

- The encodings in this description are named to match the encodings of EOR, EORS (predicates).
- The description of EOR, EORS (predicates) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \text{Pm} & 0 & 1 & \text{Pg} & 1 & \text{Pn} & 0 & \text{Pd} \\
\end{array}
\]

NOTS \(<\text{Pd}>.B, <\text{Pg}>/Z, <\text{Pn}>.B\)

is equivalent to

EORS \(<\text{Pd}>.B, <\text{Pg}>/Z, <\text{Pn}>.B, <\text{Pg}>.B\)

and is the preferred disassembly when \(\text{Pm} == \text{Pg}\).

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.

Operation

The description of EOR, EORS (predicates) gives the operational pseudocode for this instruction.
ORN (immediate)

Bitwise inclusive OR with inverted immediate (unpredicated).

Bitwise inclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

```
ORN <Zdn>.<T>, <Zdn>.<T>, #<const>
```

is equivalent to

```
ORN <Zdn>.<T>, <Zdn>.<T>, #(<const> - 1)
```

**Assembler Symbols**

- `<Zdn>` is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
- `<T>` is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<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**

The description of ORR (immediate) gives the operational pseudocode for this instruction.

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of 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 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 0 1 1 0 0 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 1 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 0 1 1 0 0 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
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 OR (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```
**ORR, ORRS (predicates)**

Bitwise inclusive OR predicate.

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the aliases **MOV (unpredicated)**, and **MOV (predicate, unpredicated)**.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags.

### Not setting the condition flags

```
0 0 1 0 0 1 1 0 0 0 1 0 1 1 0 0
Pm 0 1 Pg 0 Pn 0 Pd
```

ORR \(<Pd>\).B, \(<Pg>/Z, <Pn>\).B, \(<Pm>\).B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

### Setting the condition flags

```
0 0 1 0 0 1 1 1 0 0 1 0 1 1 0 0
Pm 0 1 Pg 0 Pn 0 Pd
```

ORRS \(<Pd>\).B, \(<Pg>/Z, <Pn>\).B, \(<Pm>\).B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

### Assembler Symbols

\(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.

\(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.

\(<Pm>\) Is the name of the second source scalable predicate register, encoded in the "Pm" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (unpredicated)</td>
<td>( S == '1' &amp;&amp; Pn == Pm &amp;&amp; Pm == Pg )</td>
</tr>
<tr>
<td>MOV (predicate, unpredicated)</td>
<td>( S == '0' &amp;&amp; Pn == Pm &amp;&amp; Pm == Pg )</td>
</tr>
</tbody>
</table>
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;
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 |
|-------------------------------------------------|-------------------------------------------------|-------------------------------------------------|
|      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
| 0    | 0    | 0    | 0    | 1    | 0    | 0    | 1    | 1    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |

**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

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ORR (immediate)

Bitwise inclusive OR with immediate (unpredicated).

Bitwise inclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction ORN (immediate).

```
0 0 0 0 0 0 1 0 1 0 0 0 0 0 ime13
```

ORR <Zdn>.<T>, <Zdn>.<T>, #<const>

```
if !HaveSVE() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>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

```
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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ORR (vectors, unpredicated)

Bitwise inclusive OR vectors (unpredicated).

Bitwise inclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the first in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (vector, unpredicated).

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;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
0 0 0 0 0 1 0 0 | size | 0 1 1 0 0 0 0 1 | Pg | Zn | Vd
```

```
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
  ifElemP[mask, e, esize] == '1' then
    result = result OR Elem[operand, e, esize];
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PFALSE

Set all predicate elements to false.
Set all elements in the destination predicate to false.

\[\begin{array}{cccccccccccccccccccc}
\end{array}\]

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);

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 0 1 0 1 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn] = result;
```

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;
```
PNEXT

Find next active predicate.

An instruction used to construct a loop which iterates over all active elements in a predicate. If all source predicate elements are false it sets the first active predicate element in the destination predicate to true. Otherwise it determines the next active predicate element following the last true source predicate element, and if one is found sets the corresponding destination predicate element to true. All other destination predicate elements are set to false. Sets the 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 0 1 1 0 0 1 1 1 0 0 1 0 Pg 0 Pdn
```

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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

| 31 30 29 28 27 26 25 24 23 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 | imm5 | 1 1 | Pg | Zn | 0 | prfop |

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

| 31 30 29 28 27 26 25 24 23 22 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 | imm5 | 1 1 | Pg | Zn | 0 | prfop |

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

The <prfop> is the prefetch operation specifier, encoded in "prfop":

---

**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. **Assembler Symbols**

The <prfop> is the prefetch operation specifier, encoded in "prfop":
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P) mask = P[g];
bits(VL) base;
bits(64) addr;
base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFB (scalar plus immediate)

Contiguous prefetch bytes (immediate index).

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|imm6|Pg|Rn|0|prfop|
```

PRFB <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = SInt(imm6);
```

Assembler Symbols

- `<prfop>`: Is the prefetch operation specifier, encoded in “prfop”:

<table>
<thead>
<tr>
<th>prfop</th>
<th>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);
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.

```
31 30 29 28 27 26 25 24 23 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 1 1 0 0 0 0 0 0 Rm 1 1 0 Pg Rn 0 prfop
```

PRFB <prfop>, <Pg>, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' 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>
<th>0000</th>
<th>PLDL1KEEP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P) mask = P[g];
bits(64) base;
bits(64) offset = X[m];
bits(64) addr;

if n == 31 then
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + (UInt(offset) << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
  offset = offset + 1;
**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 |
|-------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 0 0 0 0 1 0 0 0 | xs | 1 | Zm | 0 | 0 | 0 | Pg | Rn | 0 | prfop | msz<1> | msz<0> |
```

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 | xs | 1 | Zm | 0 | 0 | 0 | Pg | Rn | 0 | prfop | msz<1> | msz<0> |
```

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 | 1 | Zm | 1 | 0 | 0 | Pg | Rn | 0 | prfop | msz<1> | msz<0> |
```

PRFB (scalar plus vector)
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

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 0 0 0 1 0 1 1 0 0 | imm5 | 1 1 1 |Pg| Zn| 0 | prfop |

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

| 31 30 29 28 27 26 25 24 23 22 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 0 0 | imm5 | 1 1 1 | Pg| Zn| 0 | prfop |

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”:
<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 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];
bins(VL) base;
bits(64) addr;
base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFD (scalar plus immediate)

Contiguous prefetch doublewords (immediate index).

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 1 imm6 0 1 1 Pg Rn 0 prfop</td>
</tr>
</tbody>
</table>

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

<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);
**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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

msz<1>msz<0>

PRFD <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == "11111" then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;

**Assembler Symbols**

<prfop>  Is the prefetch operation specifier, encoded in “prfop”:

<table>
<thead>
<tr>
<th>prfop</th>
<th>prfop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

\texttt{CheckSVEEnabled();}
\texttt{integer elements = VL \text{DIV} esize;}
\texttt{bits(PL) mask = P[g];}
\texttt{bits(64) base;}
\texttt{bits(64) offset = X[m];}
\texttt{bits(64) addr;}

\texttt{if n == 31 then}
\texttt{\hspace{1em}base = SP[;]}
\texttt{else}
\texttt{\hspace{1em}base = X[n];}

\texttt{for e = 0 to elements-1}
\texttt{\hspace{1em}if \texttt{ElemP[mask, e, esize] == '1' then}
\texttt{\hspace{2em}addr = base + (\texttt{UInt}(offset) \text{<<} scale);
\texttt{\hspace{2em}}\texttt{Hint_Prefetch(addr, pref_hint, level, stream);
\texttt{\hspace{2em}}offset = offset + 1;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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

```
|31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0|
|1 0 0 0 0 1 0 0 0 |xs| 1 | Zm | 0 | 1 | 1 | Pg | Rn | 0 | prfop|
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;

32-bit unpacked scaled offset

```
|31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0|
|1 1 0 0 0 1 0 0 0 |xs| 1 | Zm | 0 | 1 | 1 | Pg | Rn | 0 | prfop|
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;

64-bit scaled offset

```
|31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0|
|1 1 0 0 0 1 0 0 0 1 1 | Zm | 1 | 1 | 1 | Pg | Rn | 0 | prfop|
```

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#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();
iconteger 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);
**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 | 1 | Pg | Zn | 0 | prfop |

msz<1>msz<0>
```

PRFH `<prfop>`, `<Pg>`, `[<Zn>.S{, #<imm>}]`

if `!HaveSVE()` then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);

### 64-bit element

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | imm5 | 1 | 1 | 1 | Pg | Zn | 0 | prfop |

msz<1>msz<0>
```

PRFH `<prfop>`, `<Pg>`, `[<Zn>.D{, #<imm>}]`

if `!HaveSVE()` then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);

### Assembler Symbols

`<prfop>` Is the prefetch operation specifier, encoded in “prfop”:
<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>

**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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**PRFH (scalar plus immediate)**

Contiguous prefetch halfwords (immediate index).

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

![Memory address layout](image)

**Assembler Symbols**

- `<prfop>` Is the prefetch operation specifier, encoded in "prfop":
  
<table>
<thead>
<tr>
<th>prfop</th>
<th>prfop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- `<imm>` Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) addr;

if n == 31 then
  base = SP[];
else
  base = X[n];
addr = base + ((offset * elements) << scale);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Hint_Prefetch(addr, pref_hint, level, stream);
  addr = addr + (1 << scale);
**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 0 0 0 1 | 0 0 1 0 0 0 0 0 0 | Rm | 1 1 0 | Pg | Rn | 0 | prfop |

msz<1>msz<0>

PRFH <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;

**Assembler Symbols**

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

<table>
<thead>
<tr>
<th>&lt;prfop&gt;</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PDL3STRM</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
  ifElemP[mask, e, esize] == '1' then
    addr = base + (UInt(offset) << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
  offset = offset + 1;

**PRFH (scalar plus vector)**

Gather prefetch halfwords (scalar plus vector).

Gather prefetch of halfwords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 2. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: **32-bit scaled offset**, **32-bit unpacked scaled offset** and **64-bit scaled offset**

### 32-bit scaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 1 0 0 0 1 0 0 0 xs 1 Zm 0 0 1 Pg Rn 0 prfop |
```

msz<1>msz<0>

```plaintext
PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;

### 32-bit unpacked scaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 1 1 0 0 1 0 0 0 xs 1 Zm 0 0 1 Pg Rn 0 prfop |
```

msz<1>msz<0>

```plaintext
PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;

### 64-bit scaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 1 1 0 0 0 1 0 0 1 1 Zm 1 0 1 Pg Rn 0 prfop |
```

msz<1>msz<0>
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>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm>  Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod>  Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) addr;
bits(VL) offset;
if n == 31 then
  base = SP[
else
  base = X[n];
  offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize] << offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
PRFW (vector plus immediate)

Gather prefetch words (vector plus immediate).

Gather prefetch of words from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive addresses are not prefetched from memory.
The `<prfop>` symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 1  | 1  | 1  | Pg | Zn | 0  | prfop |
```

```
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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | Pg | Zn | 0  | prfop |
```

```
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>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>

&lt;Pg&gt; Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

&lt;Zn&gt; Is the name of the base scalable vector register, encoded in the “Zn” field.

&lt;imm&gt; 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);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFW (scalar plus immediate)

Contiguous prefetch words (immediate index).

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

![Hexadecimal diagram]

PRFW <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```plaintext
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></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

```
void 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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 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>Pg</th>
<th>Rn</th>
<th>prfop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

**PRFW**<prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```plaintext
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>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.
- `<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 v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFW (scalar plus vector)

Gather prefetch words (scalar plus vector).

Gather prefetch of words from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 4. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: **32-bit scaled offset**, **32-bit unpacked scaled offset** and **64-bit scaled offset**

### 32-bit scaled offset

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 0 0 0 0 1 0 0 0 | Zm | 0 | 1 | 0 | Pg | Rn | 0 | prfop
```

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 | Zm | 0 | 1 | 0 | Pg | Rn | 0 | prfop
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;

### 64-bit scaled offset

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 0 0 1 1 | Zm | 1 | 1 | 0 | Pg | Rn | 0 | prfop
```

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
inger integer esize = 64;
inger integer g = UInt(Pg);
inger integer n = UInt(Rn);
inger integer m = UInt(Zm);
inger integer level = UInt(prfop<2:1>);
inger boolean stream = (prfop<0> == '1');
inger pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
inger integer offs_size = 64;
inger boolean offs_unsigned = TRUE;
inger 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();
inger integer elements = VL DIV esize;
inger bits(PL) mask = P[g];
inger bits(64) base;
inger bits(64) addr;
inger bits(VL) offset;

if n == 31 then
    base = SP[];
else
    base = X[n];
offset = Z[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
Set condition flags for predicate.

Sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate source register, and the V flag to zero.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 | 1 0 0 1 0 | 0 1 0 1 0 1 0 0 0 1 1 | Pg | 0 | Pn | 0 | 0 | 0 | 0 | 0

PTEST <Pg>, <Pn>.B

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();
bite(PL) mask = P[g];
bite(PL) result = P[n];
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
**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

<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></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></td>
<td>pattern</td>
<td>0</td>
<td>Pd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<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></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>0</td>
<td>0</td>
<td></td>
<td>pattern</td>
<td>0</td>
<td>Pd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PTRUES <Pd>.<T>{, <pattern>}

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = TRUE;
bits(5) pat = pattern;

### Assembler Symbols

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<pattern>** Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
### 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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

<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>
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**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1 0 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

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;
RBIT

Reverse bits (predicated).

Reverse bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|---------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 0 0 0 0 1 0 1 | 1 0 0 1 1 1 0 0 | Pg | Zn | Zd |
```

RBIT <Zd>,<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = BitReverse(element);
Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| S  |

RDFFR <Pd>.B

if !HaveSVE() then UNDEFINED;
integer d = UInt(Pd);

Asmmebler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

CheckSVEEnabled();
bias(PL) ffr = FFR[];
P[d] = ffr;
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>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>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|<Pd>|<Pg>|S|

RDFFR <Pd>.B, <Pg>/Z

if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

<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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|<Pd>|<Pg>|S|

RDFFRS <Pd>.B, <Pg>/Z

if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) ffr = FFR[];
bits(PL) result = ffr AND mask;
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d] = result;
**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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 1 1 1 1 1 1 0 1 0 1 0 0 1 1 1 1 1 1 0 1 1 0</td>
</tr>
</tbody>
</table>

RDVL \(<Xd\), \#<imm>\)

```plaintext
if !HaveSVE() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

**Assembler Symbols**

- \(<Xd>\): Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
- \(<imm>\): Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer len = imm * (VL DIV 8);
X[d] = len<63:0>;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 | 0 | 1 | 0 | 1 | size | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Pn | 0 | Pd |

REV <Pd>.<T>, <Pn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer d = UInt(Pd);

**Assembler Symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

CheckSVEEnabled();
bits(PL) operand = P[n];
bits(PL) result = Reverse(operand, esize DIV 8);
P[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
REV (vector)

Reverse all elements in a vector (unpredicated).

Reverse the order of all elements in the source vector and place in the destination vector. This instruction is unpredicated.

REV <Zd>..<T>, <Zn>..<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
bits(VL) operand = Z[n];
bits(VL) result = Reverse(operand, esize);
Z[d] = result;
REVB, REVH, REVW

Reverse bytes / halfwords / words within elements (predicated).

Reverse the order of 8-bit bytes, 16-bit halfwords or 32-bit words within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: Byte, Halfword and Word

**Byte**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 1 0 0 1 0 0 | Pg | Zn | Zd |

REVB <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 8;

**Halfword**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 1 0 1 1 0 0 | Pg | Zn | Zd |

REVH <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 16;

**Word**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 1 1 0 1 0 0 | Pg | Zn | Zd |

REVW <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 32;

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in "size":

Page 2043
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**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = Reverse(element, swsize);

Z[d] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SABD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer absdiff = Abs(element1 - element2);
        Elem[result, e, esize] = absdiff<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**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  1  0  0  size  0  0  0  0  0  0  1  Pg  Zn  Vd |
+-------------------+-------------------+-------------------+-------------------+
```

`SADDV <Dd>, <Pg>, <Zn>;//`

```
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>;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 1 0 1 0 1 | Pg     Zn     Zd

int_U
```

SCVTF <Zd>.H, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to half-precision

```
0 1 1 0 0 1 0 1 | 0 1 0 1 0 1 0 1 | Pg     Zn     Zd

int_U
```

SCVTF <Zd>.H, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to single-precision

```
0 1 1 0 0 1 0 1 | 1 0 0 1 0 1 0 1 | Pg     Zn     Zd

int_U
```
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

---

32-bit to double-precision

---

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 = FPRoundingMode(FPCR[]);

---

64-bit to half-precision

---

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 = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

---

64-bit to single-precision

---

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

<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>
</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>0</td>
</tr>
</tbody>
</table>

int_U

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(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding);
        Elem[result, e, esize] = ZeroExtend(fpval);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SDIV

Signed divide (predicated).

Signed divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

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>;
    end if
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
  end if
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

The MOVPRFX instruction must specify the same destination register as this instruction.

The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SDIVR

Signed reversed divide (predicated).

Signed reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

R U

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SDOT (vectors)

Signed integer dot product.

The signed integer dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

This instruction is unpredicated.

\[
\begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SDOT (indexed)

Signed integer indexed dot product.

The signed integer indexed dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

31 30 29 28 27 26 25 24 23 22 21 20 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 0 0 0 Zn Zda

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 1 1 i1 Zm 0 0 0 0 0 0 Zn Zda

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the 32-bit variant: is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.
For the 64-bit variant: is the immediate index of a quadtuplet of four 16-bit elements within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.
Operation

**CheckSVEEnabled();**

integer elements = \texttt{VL} \texttt{DIV} esize;
integer eltspersegment = 128 \texttt{DIV} 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;

for e = 0 to elements-1
    integer segmentbase = e - (e \texttt{MOD} eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = \texttt{Elem}[operand3, e, esize];
    for i = 0 to 3
        integer element1 = \texttt{SInt}(\texttt{Elem}[operand1, 4 * e + i, esize \texttt{DIV} 4]);
        integer element2 = \texttt{SInt}(\texttt{Elem}[operand2, 4 * s + i, esize \texttt{DIV} 4]);
        \texttt{res} = \texttt{res} + element1 * element2;
    \texttt{Elem}[result, e, esize] = \texttt{res};

\texttt{Z}[da] = result;

Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is \texttt{UNPREDICTABLE}:

- The \texttt{MOVPRFX} instruction must be unpredicated.
- The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SEL (predicates)

Conditionally select elements from two predicates.

Read active elements from the first source predicate and inactive elements from the second source predicate and place in the corresponding elements of the destination predicate. Does not set the condition flags.

This instruction is used by the alias MOV (predicate, predicated, merging).

<table>
<thead>
<tr>
<th>&lt;Pd&gt;</th>
<th>&lt;Pg&gt;</th>
<th>&lt;Pn&gt;</th>
<th>&lt;Pm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SEL <Pd>.B, <Pg>, <Pn>.B, <Pm>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (predicate, predicated, merging)</td>
<td>Pd == Pm</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1;
    else
        ElemP[result, e, esize] = element2;
P[d] = result;
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).

SEL <Zd>.<T>, <Pg>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector, predicated)</td>
<td>Zd == Zm</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1;
    else
        Elem[result, e, esize] = element2;
Z[d] = result;
SETFFR

Initialise the first-fault register to all true.

Initialise the first-fault register (FFR) to all true prior to a sequence of first-fault or non-fault loads. This instruction is unpredicated.

![Binary representation of initial FFR state]

```
0 0 1 0 0 1 0 1 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```c
if !HaveSVE() then UNDEFINED;
```

**Operation**

```c
CheckSVEEnabled();
FFR[] = Ones(PL);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMAX (vectors)

Signed maximum vectors (predicated).

Determine the signed maximum of active elements of the second source vector and corresponding elements of the first
source vector and destructively place the results in the corresponding elements of the first source vector. Inactive
elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 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  |

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMAX (immediate)

Signed maximum with immediate (unpredicated).

Determine the signed maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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>

SMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer size = 8 << UInt(size);
d 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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMAXV

Signed maximum reduction to scalar.

Signed maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 0 1 0 0 0</td>
</tr>
</tbody>
</table>

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>;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | Pg | Zm | Zdn |

SMIN \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>\)

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})\);
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 = \text{VL} \div \text{esize};
bits(PL) mask = \text{P}[g];
bits(VL) operand1 = \text{Z}[dn];
bits(VL) operand2 = \text{Z}[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = \text{Int(Elem[operand1, e, esize], unsigned)};
  integer element2 = \text{Int(Elem[operand2, e, esize], unsigned)};
  if \text{ElemP[mask, e, esize]} == '1' then
    integer minimum = \text{Min(element1, element2)};
    \text{Elem[result, e, esize]} = minimum<esize-1:0>;
  else
    \text{Elem[result, e, esize]} = \text{Elem[operand1, e, esize]};
\Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMIN (immediate)

Signed minimum with immediate (unpredicated).

Determine the signed minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 0 0 1 0 1 | size | 1 0 1 0 1 0 1 1 0 | imm8 | Zdn |

SMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);

Assembler Symbols

- <Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- <T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- <imm> Is the signed immediate operand, in the range -128 to 127, encoded in the “imm8” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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 1 0 0 1 0 1 0 0 0 1                  | Pg | Zn | Vd |

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>;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMULH

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  |

SMULH <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 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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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 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>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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SPLICE <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);

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 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[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQADD (immediate)**

Signed saturating add immediate (unpredicated).

Signed saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is “#<imm8>, LSL #8”. However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as “#0, LSL #8”.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size | 1 0 0 1 0 0 | 1 0 | 0 | 1 1 | sh | imm8 | Zdn |

SQADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;

### Assembler Symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>` Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th><code>&lt;shift&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

### Operation

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);
Z[dn] = result;

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQADD (vectors)**

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Zm | 0  | 0  | 0  | 0  | 0  | Zn | 0  | 0  | Zd | U |

**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 = \(\text{VL} \div \text{esize}\);

bits(\text{VL}) \text{operand1} = \text{Z}[n];

bits(\text{VL}) \text{operand2} = \text{Z}[m];

bits(\text{VL}) \text{result};

for e = 0 to elements-1

integer element1 = \(\text{Int(Elem[operand1, e, esize], unsigned)}\);

integer element2 = \(\text{Int(Elem[operand2, e, esize], unsigned)}\);

(Elem[\text{result, e, esize}, -]) = \text{SatQ(element1 + element2, esize, unsigned)};

\text{Z}[d] = \text{result};
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  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | imm4 | 1  | 1  | 1  | 1  | 1  | 0  | pattern | Rdn |

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  | 0  | 0  | 1  | 0  | imm4 | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | pattern | Rdn |

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”:  

<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>10001</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>MUL4</td>
</tr>
<tr>
<td>11100</td>
<td>MUL3</td>
</tr>
<tr>
<td>11110</td>
<td>MUL4</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDECD (scalar)**

Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 0 1 1 1 0 | immm | pattern | Rdn |
|---|---|---|---|

SQDECD <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer size = 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 1 0 0 1 1 1 1 | immm | pattern | Rdn |
|---|---|---|---|

SQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer size = 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>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>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 count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECD (vector)

Signed saturating decrement vector by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>Zdn</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>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>

SQDECD <Zdn>.D{, <pattern>}{, MUL #<imm>} where

- <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>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

- <imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| size<1>|size<0> | sf | D | U |

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**

| 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 | 20 | 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>|size<0> | sf | D | U |

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”: 
<table>
<thead>
<tr>
<th>pattern</th>
<th>\text{&lt;pattern&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>\text{POW2}</td>
</tr>
<tr>
<td>00001</td>
<td>\text{VL1}</td>
</tr>
<tr>
<td>00010</td>
<td>\text{VL2}</td>
</tr>
<tr>
<td>00011</td>
<td>\text{VL3}</td>
</tr>
<tr>
<td>00100</td>
<td>\text{VL4}</td>
</tr>
<tr>
<td>00101</td>
<td>\text{VL5}</td>
</tr>
<tr>
<td>00110</td>
<td>\text{VL6}</td>
</tr>
<tr>
<td>00111</td>
<td>\text{VL7}</td>
</tr>
<tr>
<td>01000</td>
<td>\text{VL8}</td>
</tr>
<tr>
<td>01001</td>
<td>\text{VL9}</td>
</tr>
<tr>
<td>01010</td>
<td>\text{VL10}</td>
</tr>
<tr>
<td>01011</td>
<td>\text{VL11}</td>
</tr>
<tr>
<td>01100</td>
<td>\text{VL12}</td>
</tr>
<tr>
<td>01101</td>
<td>\text{VL13}</td>
</tr>
<tr>
<td>01110</td>
<td>\text{VL14}</td>
</tr>
<tr>
<td>01111</td>
<td>\text{VL15}</td>
</tr>
<tr>
<td>10100</td>
<td>\text{VL16}</td>
</tr>
<tr>
<td>10101</td>
<td>\text{VL17}</td>
</tr>
<tr>
<td>10110</td>
<td>\text{VL18}</td>
</tr>
<tr>
<td>10111</td>
<td>\text{VL19}</td>
</tr>
<tr>
<td>11000</td>
<td>\text{VL20}</td>
</tr>
<tr>
<td>11001</td>
<td>\text{VL21}</td>
</tr>
<tr>
<td>11010</td>
<td>\text{VL22}</td>
</tr>
<tr>
<td>11011</td>
<td>\text{VL23}</td>
</tr>
<tr>
<td>11100</td>
<td>\text{VL24}</td>
</tr>
<tr>
<td>11101</td>
<td>\text{VL25}</td>
</tr>
<tr>
<td>11110</td>
<td>\text{VL26}</td>
</tr>
<tr>
<td>11111</td>
<td>\text{VL27}</td>
</tr>
</tbody>
</table>

<\text{imm}> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

```c
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>01110</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>1xx01</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

**<imm>**  
Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDECP (scalar)

Signed saturating decrement scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | size 1 0 1 0 1 0 1 0 0 0 1 0 0 | Pm | Rdn

SQDECP <Xdn>, <Pm>,<T>, <Wdn>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | size 1 0 1 0 1 0 1 0 0 0 1 1 0 | Pm | Rdn

SQDECP <Xdn>, <Pm>,<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Pm> Is the name of the source scalable predicate register, encoded in the “Pm” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
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'
    count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element - count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
SQDECP (vector)

Signed saturating decrement vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SQDECP <Zdn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << Uint(size);
integer m = Uint(Pm);
integer dn = Uint(Zdn);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;
for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDECW (scalar)

Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

<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>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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size<1> size<0> sf D U

SQDECW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>imm4</td>
<td>1</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>
<td></td>
</tr>
</tbody>
</table>

size<1> size<0> sf D U

SQDECW <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</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 count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECW (vector)

Signed saturating decrement vector by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

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 1 0 0 1</td>
</tr>
<tr>
<td>size&lt;1&gt; size&lt;0&gt; D U</td>
</tr>
</tbody>
</table>

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”:

```
pattern   <pattern>
00000     POW2
00001     VL1
00010     VL2
00011     VL3
00100     VL4
00101     VL5
00110     VL6
00111     VL7
01000     VL8
01010     VL16
01011     VL32
01100     VL64
01101     VL128
01110     VL256
0111x     #uimm5
101x1      #uimm5
10110      #uimm5
1x0x1      #uimm5
1x010      #uimm5
1xx00      #uimm5
11101      MUL4
11110      MUL3
11111      ALL
```

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | imm4 | 1  | 1  | 1  | 1  | 0  | 0  |

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>sf</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
</table>

SQINCB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

If !HaveSVE() then UNDEFINED;

```c
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  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | imm4 | 1  | 1  | 1  | 1  | 0  | 0  |

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>sf</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
</table>

SQINCB <Xdn>{, <pattern>{, MUL #<imm>}}
```

If !HaveSVE() then UNDEFINED;

```c
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

### Assembler Symbols

- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

SQINCB
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);
```
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 |

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 1 0 0 1 1 1 0 imm4 1 1 1 1 0 0 pattern Rdn |

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>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</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 count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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.

Unsaturated <Zdn> size<1> size<0> D U

<table>
<thead>
<tr>
<th>Zdn</th>
<th>Pattern</th>
<th>imm4</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>VL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>VL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>VL4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>VL5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>VL6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>VL7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>VL8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>VL9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>VL10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>VL11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>VL12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x0x0</td>
<td>#uimm5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assembler Symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

```

<table>
<thead>
<tr>
<th>Pattern</th>
<th>&lt;Pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00010</td>
<td>VL1</td>
</tr>
<tr>
<td>00011</td>
<td>VL2</td>
</tr>
<tr>
<td>00100</td>
<td>VL3</td>
</tr>
<tr>
<td>00101</td>
<td>VL4</td>
</tr>
<tr>
<td>00110</td>
<td>VL5</td>
</tr>
<tr>
<td>00111</td>
<td>VL6</td>
</tr>
<tr>
<td>01000</td>
<td>VL7</td>
</tr>
<tr>
<td>01001</td>
<td>VL8</td>
</tr>
<tr>
<td>01010</td>
<td>VL9</td>
</tr>
<tr>
<td>01011</td>
<td>VL10</td>
</tr>
<tr>
<td>01100</td>
<td>VL11</td>
</tr>
<tr>
<td>01101</td>
<td>VL12</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

\[
\text{CheckSVEEnabled}();
\text{integer elements = } VL \text{ DIV esize};
\text{integer count = } \text{DecodePredCount}(\text{pat}, \text{esize});
\text{bits}(VL) \text{ operand1 = } Z[dn];
\text{bits}(VL) \text{ result};
\]

\[
\text{for } e = 0 \text{ to elements-1}
\text{integer element1 = } \text{Int}([\text{operand1}, e, \text{esize}], \text{unsigned});
(\text{Elem}[\text{result}, e, \text{esize}], -) = \text{SatQ}((\text{element1} + (\text{count} * \text{imm}), \text{esize}, \text{unsigned});
\]

\[
Z[dn] = \text{result};
\]

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQINCH (scalar)

Signed saturating increment scalar by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | | | | | | | | | | | | | | | | | | | | |
|    |    |    |    |    | imm4 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

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  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | | | | | | | | | | | | | | | | | | | | | | |
|    |    |    |    |    | imm4 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

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”:
<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>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 count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

Unsaturated saturating increment vector by multiple of 16-bit predicate constraint element count.

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 0 0 0 0 pattern Zdn
```

SQINCH <Zdn>.H(, <pattern>{, MUL #<imm>})

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bites(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>&lt;pattern&gt;</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>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01010</td>
<td>VL16</td>
</tr>
<tr>
<td>01011</td>
<td>VL32</td>
</tr>
<tr>
<td>01101</td>
<td>VL128</td>
</tr>
<tr>
<td>01100</td>
<td>VL256</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQINCP (scalar)

Signed saturating increment scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|---------------|---------------|
| 0 0 1 0 0 1 0 1 | size 1 0 1 0 | 0 0 1 0 0 1 0 0 | Pm | Rdn |

SQINCP <Xdn>, <Pm>.<T>, <Wdn>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer m = Uint(Pm);
integer dn = Uint(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;

### 64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|---------------|---------------|
| 0 0 1 0 0 1 0 1 | size 1 0 1 0 | 0 0 1 0 0 1 1 0 | Pm | Rdn |

SQINCP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer m = Uint(Pm);
integer dn = Uint(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;

#### Assembler Symbols

- **<Xdn>** Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- **<Pm>** Is the name of the source scalable predicate register, encoded in the "Pm" field.
- **<T>** Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Wdn>** Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1'
    count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
**SQINCP (vector)**

Signed saturating increment vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

| 31 30 29 28 27 26 25 24 23 22 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 0 0 0 0 0 0 |

SQINCP \(<Zdn>.<T>, <Pm>.<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQINCW (scalar)

Signed saturating increment scalar by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 1 0 0 1 0 imm4 1 1 1 1 0 0 pattern Rdn

size<1>size<0> sf D U

SQINCW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 1 0 0 1 0 imm4 1 1 1 1 0 0 pattern Rdn

size<1>size<0> sf D U

SQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

SQINCW (scalar)
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>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>MUL4</td>
</tr>
<tr>
<td>11010</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQINCW (vector)**

Signed saturating increment vector by multiple of 32-bit predicate constraint element count.

 Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>pattern</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if ! HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01010</td>
<td>VL16</td>
</tr>
<tr>
<td>01011</td>
<td>VL32</td>
</tr>
<tr>
<td>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>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx0x0</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSUB (immediate)**

Signed saturating subtract immediate (unpredicated).

Signed saturating subtract of an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)}-1$. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

### Assembler Symbols

- `<Zdn>`: Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>`: Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>`: Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], ·) = SatQ(element1 - imm, esize, unsigned);
Z[dn] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQSUB (vectors)

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 0 0 1 0 0 | size | 1 | Zm | 0 0 0 1 1 0 | Zn | Zd
```

SQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;

Assembler Symbols

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);

Z[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST1B (vector plus immediate)

Scatter store bytes from a vector (immediate index).

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a vector
base plus immediate index. The index is in the range 0 to 31. Inactive elements are not written to memory.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | imm5 | 1  | 0  | 1  | Pg  | Zn  | Zt  |
| msz<1> | msz<0> |

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | imm5 | 1  | 0  | 1  | Pg  | Zn  | Zt  |
| msz<1> | msz<0> |

ST1B { <Zt>.D }, <Pg>, [<Zn>.D{, #imm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the “Zn” field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the “imm5” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
      addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
      Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;

ST1B (scalar plus immediate)

Contiguous store bytes from vector (immediate index).

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0 0 0 0 0 0 1 1 1</td>
</tr>
</tbody>
</table>

```
STIB { <Zt>,<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
  addr = addr + mbytes;
```

ST1B (scalar plus immediate)
**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 |

ST1B { <Zt>,<T>, [Xn|SP>, Xm}]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 8;

**Assembler Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
    base = SP[1];
else
    base = X[n];

for e = 0 to elements-1
addr = base + UInt(offset) * mbytes;
if Elem[ 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

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>P</td>
<td>g</td>
<td>R</td>
<td>n</td>
<td>Z</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{msz} < 1 > \text{msz} < 0 >\]

```c
ST1B \{ <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 = 8;
integer offs_size = 32;
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>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>Z</td>
<td>m</td>
<td>1</td>
<td>x</td>
<td>s</td>
<td>0</td>
<td>P</td>
<td>g</td>
<td>R</td>
<td>n</td>
<td>Z</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{msz} < 1 > \text{msz} < 0 >\]

```c
ST1B \{ <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 esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 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>Z</td>
<td>m</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>g</td>
<td>R</td>
<td>n</td>
<td>Z</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{msz} < 1 > \text{msz} < 0 >\]
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

**Assembler Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn>|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(VL) offset = Z[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
   ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    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 v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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  | 0  | imm5| 1  | 0  | 1  | Pg | Zn | Zt | msz<1>|msz<0>|msz<1>|msz<0>|

ST1D {<Zt>.D}, <Pg>, [<Zn>.D{, #imm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST1D (scalar plus immediate)

Contiguous store doublewords from vector (immediate index).

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

```
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | imm4 | 1 | 1 | Pg | Rn | Zt |
```

msz<1>msz<0>

ST1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 64;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
```

```
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
    addr = addr + mbytes;
```
ST1D (scalar plus scalar)

Contiguous store doublewords from vector (scalar index).

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | Rm | 0 | 1 | 0 | Pg | Rn | Zt |
```

ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
  offset = offset + 1;
ST1D (scalar plus vector)

Scatter store doublewords from a vector (vector index).

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements are not written to memory.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

### 32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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>
<th></th>
<th></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

```
ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 3;

### 32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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>
<th></th>
<th></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

```
ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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>
<th></th>
<th></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

```
ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>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>
<td></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.
- **<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 = \texttt{VL} \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 = msize \div 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = \texttt{SP}[];
else
    base = \texttt{X}[n];
for e = 0 to elements-1
    if \texttt{Elem}[mask, e, esize] == '1' then
        integer off = \texttt{Int}([\texttt{Elem}[offset, e, esize]<offs\_size-1:0>, offs\_unsigned]);
        addr = base + (off \ll 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

<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>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>imm5</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>Pg</td>
</tr>
<tr>
<td>21</td>
<td>Zn</td>
</tr>
<tr>
<td>20</td>
<td>Zt</td>
</tr>
<tr>
<td>19</td>
<td>msz&lt;1&gt;</td>
</tr>
<tr>
<td>18</td>
<td>msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

STIH { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offset = UInt(imm5);

### 64-bit element

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>imm5</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>Pg</td>
</tr>
<tr>
<td>21</td>
<td>Zn</td>
</tr>
<tr>
<td>20</td>
<td>Zt</td>
</tr>
<tr>
<td>19</td>
<td>msz&lt;1&gt;</td>
</tr>
<tr>
<td>18</td>
<td>msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

STIH { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offset = UInt(imm5);

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | size | 0 | imm4 | 1 | 1 | 1 | Pg | Rn | Zt |
```

STIH { <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.
- `<T>` Is the 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
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
  addr = addr + mbytes;
```
**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  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

ST1H {<Zt>..<T>}, <Pg>, [<Xn|SP>], <Xm>, LSL #1

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 16;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];

for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
        offset = offset + 1;
**ST1H (scalar plus vector)**

Scatter store halfwords from a vector (vector index).

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements are not written to memory.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

### 32-bit scaled offset

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[1 1 1 0 0 1 0 0 | 1 1 1 | Zm | 1 | xs | 0 | Pg | Rn | Zt]
```

msz<1>msz<0>

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

### 32-bit unpacked scaled offset

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[1 1 1 0 0 1 0 0 | 1 0 1 | Zm | 1 | xs | 0 | Pg | Rn | Zt]
```

msz<1>msz<0>

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

### 32-bit unpacked unscaled offset

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[1 1 1 0 0 1 0 0 | 1 0 0 | Zm | 1 | xs | 0 | Pg | Rn | Zt]
```

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 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  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

msz<1>msz<0>

### 64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 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  |

msz<1>msz<0>

### 64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

msz<1>msz<0>
if `HaveSVE()` then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 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(VL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if `HaveMTEExt()` then `SetTagCheckedInstruction(TRUE)`;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        `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  | 0  | 1  | 1  |   | 0  | 1  | 1  |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| msz<1>msz<0> |

ST1W \{ <Zt>.S \}, \{<Pg>, \{<Zn>.S{}, #<imm>}\}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offset = UInt(imm5);

64-bit element

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  |   | 1  | 0  | 1  |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| msz<1>msz<0> |

ST1W \{ <Zt>.D \}, \{<Pg>, \{<Zn>.D{}, #<imm>}\}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if $ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend($Elem[base, e, esize], 64) + offset * mbytes;
    Mem[addr, mbytes, AccType_NORMAL] = $Elem[src, e, esize]<msize-1:0>;

ST1W (scalar plus immediate)

Contiguous store words from vector (immediate index).

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

```
|   31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0 |
|-------|------|------|------|------|------|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1     | 1    | 1    | 0    | 1    | 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   |

size<0> size imm4 Pg Rn Zt

msz<1> msz<0>

ST1W { <Zt>,<T> }, <Pg>, [<Xn|SP>{}<, #<imm>, MUL VL]}

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size<0>":

```
<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    addr = addr + mbytes;
    if Elem[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
        addr = addr + mbytes;
```

ST1W (scalar plus immediate)
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
```

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;

Assmblr Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
  offset = offset + 1;
ST1W (scalar plus vector)

Scatter store words from a vector (vector index).

Scatter store of words from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements are not written to memory.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

### 32-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>msz&lt;1&gt; msz&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 0 1 1 0 1 1 1</td>
<td>Zm 1</td>
</tr>
</tbody>
</table>

ST1W { <Zt>.S }, <Pg>, [<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 offs_unsigned = xs == '0';
integer scale = 2;

### 32-bit unpacked scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>msz&lt;1&gt; msz&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 0 1 1 0 1 1 1</td>
<td>Zm 1</td>
</tr>
</tbody>
</table>

ST1W { <Zt>.D }, <Pg>, [<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 offs_unsigned = xs == '0';
integer scale = 2;

### 32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>msz&lt;1&gt; msz&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 0 1 1 0 1 1 1</td>
<td>Zm 1</td>
</tr>
</tbody>
</table>

ST1W (scalar plus vector)
ST1W { <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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 1  | 0  | Zm | 1  | xs | 0  | Pg | Rn | Zt |

ST1W { <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 esize = 32;
integer msize = 32;
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  | 1  | 0  | 0  | 1  | Zm | 1  | 0  | 1  | Pg | Rn | Zt |

ST1W { <Zt>.D }, <Pg>, [<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 offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | Zm | 1  | 0  | 1  | Pg | Rn | Zt |
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;

Assembling Symbols

| <Zt> | Is the name of the scalable vector register to be transferred, encoded in the "Zt" field. |
| <Pg> | Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field. |
| <Xn|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <Zm> | Is the name of the offset scalable vector register, encoded in the "Zm" field. |
| <mod> | Is the index extend and shift specifier, encoded in "xs": |

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(VL) offset = Z[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(UNpredictable_CHECKSPNONEACTIVE) then
        CheckSAPrealign();
    base = SP[];
else
    base = X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
**ST2B (scalar plus immediate)**

Contiguous store two-byte structures from two vectors (immediate index).

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```assembly
ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}}
```

### Assembler Symbols

- `<Zt1>`: Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>`: Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

### Operation

```assembly
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
```

---

*ST2B (scalar plus immediate)*
ST2B (scalar plus scalar)

Contiguous store two-byte structures from two vectors (scalar index).

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 | 0 0 | 0 1 | Rm | 0 1 1 | Pg | Rn | Zt
msz<1>msz<0>
```

ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 2;

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if 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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>msz&lt;1&gt; msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(PL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
fors 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.

\[ \text{ST2D} \{ <Zt1>.D, <Zt2>.D \}, <Pg>, [<Xn|SP>, <Xm>, LSL #3] \]

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 = 2;

Assembler Symbols

<Zt> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAAddrment();
  else
    base = SP[];
else
  base = X[n];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
**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.

| 31 30 29 28 27 26 25 24 23 22 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 1 1 1 1 1 | imm4PgRnZt

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
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
            base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
            base = X[n];
else
    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 immediate)
ST2H (scalar plus scalar)

Contiguous store two-halfword structures from two vectors (scalar index).

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 1 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 0 1 1 0 1 1 |
```

```
ST2H { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]
```

```c
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 2;
```

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  else
    base = SP[];
else
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
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 0 1 1 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 1 1 1 1 1 |

\[ \text{msz<1>msz<0>} \]

\[ \text{ST2W \{ <Zt1>.S, <Zt2>.S \}, <Pg>, [<Xn]\{SP\}, #<imm>, MUL VL} \]

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer \(t = \text{ UInt}(Zt)\);
integer \(n = \text{ UInt}(Rn)\);
integer \(g = \text{ UInt}(Pg)\);
integer esize = 32;
integer offset = \(\text{ SInt}(\text{imm4})\);
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

\[ \text{CheckSVEEnabled()}; \]
integer elements = \(\text{ VL DIV esize}\);
bits(64) base;
bits(64) addr;
bits(PL) mask = \(P[g]\);
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if \(n == 31\) then
  if \text{LastActiveElement}(mask, esize) >= 0 ||
    \text{ConstrainUnpredictableBool}(\text{Unpredictable_CHECKSPNONEACTIVE}) then
      \text{CheckSPAlignment}();
    if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(FALSE);
      base = \(SP[\]};
  else
    if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(TRUE);
      base = \(X[n]\);
for \(r = 0\) to \(nreg-1\)
values[r] = \(Z[(t+r) MOD 32]\);
addr = base + offset * elements * nreg * mbytes;
for \(e = 0\) to \(\text{elements-1}\)
for \(r = 0\) to \(\text{nreg-1}\)
  if \(\text{ElemP[mask, e, esize] == 'l'}\) then
    \text{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.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
  ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for 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.

31 30 29 28 27 26 25 24 23 22 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 | imm4 | 1 1 1 | Pg | Rn | Zt |

msz<1>msz<0>

ST3B { <Zt1>, <Zt2>.B, <Zt3>.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 = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
        addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3B (scalar plus scalar)

Contiguous store three-byte structures from three vectors (scalar index).

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

31 30 29 28 27 26 25 24 23 22 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 | Rm | 0 1 1 | Pg | Rn | Zt
msz<1>msz<0>

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

```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 SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3D (scalar plus immediate)

Contiguous store three-doubleword structures from three vectors (immediate index).

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```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 1 0 0 1 0 1 1 1 0 1 imm4 1 1 1 Pg Rn Zt
```

```plaintext
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

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

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 SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3H (scalar plus immediate)

Contiguous store three-halfword structures from three vectors (immediate index).

Contiguous store three-halfword structures, each from the same element number in three vector registers to the
memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to
21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to
the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to
memory.

```
```

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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
31 30 29 28 27 26 25 24 23 22 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 | Rm | 0 1 1 | Pg | Rn | Zt

msz<1>msz<0>
```

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 SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
    base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
    offset = offset + nreg;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3W (scalar plus immediate)

Contiguous store three-word structures from three vectors (immediate index).

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 0 1 0 1 imm4 1 1 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>
```


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
```
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.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 | 1 | 0 | Rm | 0 1 1 | Pg | Rn | Zt
msz<1>msz<0>
```

```
ST3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 3;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        base = SP[];
else
    base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
            addr = addr + mbytes;
            offset = offset + nreg;
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.

```
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
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
ST4B (scalar plus scalar)

Contiguous store four-byte structures from four vectors (scalar index).

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

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 SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  base = SP[];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
ST4D (scalar plus immediate)

Contiguous store four-doubleword structures from four vectors (immediate index).

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 1 1 1 imm4 1 1 1 Pg Rn Zt
msz<1>msz<0>
```

```
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

\[
\begin{array}{ccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 |
\end{array}
\begin{array}{cccc}
Rm & 0 & 1 & 1 |
\end{array}
\begin{array}{cccc}
Pg & Rn & Zt |
\end{array}
\]

msz<1>msz<0>

ST4D \{ <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D \}, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer nreg = 4;

Assembler Symbols

- \(<Zt1>\) Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- \(<Zt2>\) Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- \(<Zt3>\) Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- \(<Zt4>\) Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- \(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        base = SP[];
else
    base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
            addr = addr + mbytes;
            offset = offset + nreg;
**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.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[1 1 1 0 0 1 0 0 1 1 1 \mid \text{imm4} \mid 1 1 1 \mid \text{Pg} \mid \text{Rn} \mid \text{Zt}\]
```

```
```

```java
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 = \text{VL} \text{ DIV} \text{esize};
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize \text{ DIV} 8;
array [0..3] of bits(\text{VL}) values;

if n == 31 then
  if \text{LastActiveElement}(mask, \text{esize}) >= 0 ||
      \text{ConstrainUnpredictableBool}(\text{Unpredictable_CHECKSPNONEACTIVE}) then
    \text{CheckSPAlignment}();
  else
    if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(\text{FALSE});
    base = SP[];
  end
else
  if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(\text{TRUE});
  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, \text{esize}] == '1' then
      \text{Mem}[addr, mbytes, \text{AccType_NORMAL}] = \text{Elem}[values[r], e, \text{esize}];
    addr = addr + mbytes;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST4H (scalar plus scalar)

Contiguous store four-halfword structures from four vectors (scalar index).

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 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

msz<1>msz<0>
```


if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 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

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
for 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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
**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 | Pg | Rn | Zt |

msz<1>msz<0>


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 4;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 0 1 0 1 0 1 1 0 1 1 1 1 Rm 0 1 1 1 0 1 1 Pg Rn Zt
msz<1> msz<0>
```

```
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 = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + Uint(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
    offset = offset + nreg;
```

STNT1B (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.

STNT1B {<Zt>.B}, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
      base = SP[0];
  else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
      base = X[n];
  src = Z[t];
endif
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;
  endif
endif

STNT1B (scalar plus scalar)

Contiguous store non-temporal bytes from vector (scalar index).

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0
msz<1> msz<0>

STNT1B { <Zt>.B, <Pg>, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
```

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset = X[m];
bits(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
src = Z[t];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_Stream] = Elem[src, e, esize];
    offset = offset + 1;
```
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.

```
| 31 30 29 28 27 26 25 24 23 22 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 1 imm4 | 1 1 1 Pg   |
|                 |                |   Rn Zt    |
```

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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
src = Z[t];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
  addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 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 0 | Rm 0 1 1 | Pg 0 1 1 | Rn 0 1 1 | Zt 0 1 1 |
```

STNTID { <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**

```assembly
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset = X[m];
bits(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
src = Z[t];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
    offset = offset + 1;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
[ 1 1 1 0 0 1 0 0 1 1 1 ]
msz<1>msz<0>
```

STNT1H { <Zt>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
src = Z[t];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
  addr = addr + mbytes;
STNT1H (scalar plus scalar)

Contiguous store non-temporal halfwords from vector (scalar index).

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
| 31 30 29 28 27 26 25 24 23 22 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 1 0 0 | Rm | 0 1 1 | Pg | Rn | Zt |
```

STNT1H { <Zt>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset = X[m];
bits(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
src = Z[t];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
  offset = offset + 1;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1W (scalar plus immediate)

Contiguous store non-temporal words from vector (immediate index).

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

\[
\begin{array}{cccccccccccccccccccccc}
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & \text{imm}4 & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt}
\end{array}
\]

\text{msz<1> msz<0>}

STNT1W \{ <Zt>.S }, <Pg>, \[<Xn|SP>\{, #<imm>, MUL VL}\}

\text{if !HaveSVE() then UNDEFINED;}
\text{integer } t = \text{UInt}(Zt);
\text{integer } n = \text{UInt}(Rn);
\text{integer } g = \text{UInt}(Pg);
\text{integer } esize = 32;
\text{integer } offset = \text{SInt}(\text{imm}4);

\textbf{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.

\textbf{Operation}

\texttt{CheckSVEEnabled();}
\texttt{integer elements = VL DIV esize;}
\texttt{bits(64) base;}
\texttt{bits(64) addr;}
\texttt{constant integer mbytes = esize DIV 8;}
\texttt{bits(VL) src;}
\texttt{bits(PL) mask = P[g];}

\text{if } n == 31 \text{ then}
  \text{if LastActiveElement(mask, esize) >= 0 ||}
  \text{ConstrainUnpredictableBool(Unpredictable_CHECKPNONEACTIVE) then}
    \text{CheckSPAlignment();}
  \text{if HaveMTEExt() then SetTagCheckedInstruction(FALSE);}
    \text{base = SP[];}
  \text{else}
    \text{if HaveMTEExt() then SetTagCheckedInstruction(TRUE);}
    \text{base = X[n];}
\text{src = Z[t];}

\text{addr = base + offset * elements * mbytes;}
\text{for } e = 0 \text{ to elements-1}
  \text{if ElemP[mask, e, esize] == '1' then}
    \text{Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];}
  \text{addr = addr + mbytes;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1W (scalar plus scalar)

Contiguous store non-temporal words from vector (scalar index).

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 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  |

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 HaveMTEEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
      base = SP[];
    else
      base = X[n];
else
  src = Z[t];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
    offset = offset + 1;
**STR (predicate)**

Store predicate register.

Store a predicate register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated. The store is performed as a stream of bytes containing 8 consecutive predicate bits in ascending element order, without any endian conversion.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm9h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm9l</td>
<td>Rn</td>
<td>0</td>
<td>Pt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STR \(<\text{Pt}\), [<Xn|SP>\{, \#<imm>, MUL VL}]\)

if \(\text{sve}()\) then UNDEFINED;
integer t = \(\text{UInt}(\text{Pt})\);
integer n = \(\text{UInt}(\text{Rn})\);
integer imm = \(\text{SInt}(\text{imm9h}:\text{imm9l})\);

**Assembler Symbols**

\(<\text{Pt}>\) Is the name of the scalable predicate transfer register, encoded in the "Pt" field.

\(<\text{Xn}\text{|SP}>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{imm}>\) Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

**Operation**

\(\text{CheckSVEEnabled}();\)
integer elements = \(\text{PL} \div \text{8}\);
bits(PL) src;
bits(64) base;
integer offset = imm * elements;

if \(n == 31\) then
   \(\text{CheckSPAlignment}();\)
   if \(\text{sve}()\) then \(\text{SetTagCheckedInstruction}(\text{FALSE})\);
   base = \(\text{SP}[t]\);
else
   if \(\text{sve}()\) then \(\text{SetTagCheckedInstruction}(\text{TRUE})\);
   base = \(\text{X}[n]\);

src = \(\text{P}[t]\);
boolean aligned = \(\text{AArch64.CheckAlignment}(\text{base} + \text{offset}, 2, \text{AccType_NORMAL}, \text{TRUE})\);
for \(e = 0 \text{ to elements - 1}\)
   \(\text{AArch64.MemSingle}(\text{base} + \text{offset}, 1, \text{AccType_NORMAL}, \text{aligned}) = \text{Elem}[\text{src}, e, 8];\)
offset = offset + 1;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**STR (vector)**

Store vector register.

Store a vector register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The store is performed as a stream of byte elements in ascending element order, without any endian conversion.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  |

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
    CheckS人才队伍Alignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

src = Z[t];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_NORMAL, TRUE);
for e = 0 to elements-1
    AArch64.MemSingle[base + offset, 1, AccType_NORMAL, aligned] = Elem[src, e, 8];
    offset = offset + 1;
```
SUB (vectors, predicated)

Subtract vectors (predicated).

Subtract active elements of the second source vector from corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector 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 0 0 0 0 1 0 0 Pg Zm Zdn</td>
</tr>
</tbody>
</table>

SUB <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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SUB (immediate)

Subtract immediate (unpredicated).

Subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

The instruction is encoded in the following format:

```
0 0 1 0 0 1 0 1 | size 1 0 0 0 1 1 1 sh | imm8 | 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>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>` Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 - imm;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**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.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00</th>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
SUB <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>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>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();
iconteger elements = VL DIV esize;
bhits(VL) operand1 = Z[n];
bhits(VL) operand2 = Z[m];
bhits(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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
SUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = element2 - element1;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SUBR (immediate)**

Reversed subtract from immediate (unpredicated).

Reversed subtract from an unsigned immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size | 1 0 0 0 0 1 1 1 | imm8 | sh | 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>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<imm>** Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- **<shift>** Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  Elem[result, e, esize] = (imm - element1)<esize-1:0>;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SUNPKHI, SUNPKLO

Signed unpack and extend half of vector.

Unpack elements from the lowest or highest half of the source vector and then sign-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: High half and Low half

High half

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 | size 1 1 0 0 0 1 0 0 1 1 1 0 | Zn | Zd
                  U   H
```

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 0 0 0 1 1 1 0 | Zn | Zd
                  U   H
```

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

\texttt{CheckSVEEnabled();
integer elements = \texttt{VL} \ \texttt{DIV} \ \texttt{esize};
integer hsize = \texttt{esize} \ \texttt{DIV} \ 2;
bits(\texttt{VL}) operand = \texttt{Z[n]};
bits(\texttt{VL}) result;

for \ e = 0 \ to \ elements-1
  bits(hsize) element = \texttt{if hi then E}l\texttt{em[operand, e + elements, hsize] else E}l\texttt{em[operand, e, hsize];}
  \texttt{Elem[result, e, esize] = Extend(e}lement, esize, unsigned);
\texttt{Z[d] = result;}

\text{Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12}

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

| 31 30 29 28 27 26 25 24 23 22 21 20 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 | 0 1 0 1 | Pg | Zn | Zd |

SXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;

### Halfword

| 31 30 29 28 27 26 25 24 23 22 21 20 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 1 | 0 1 0 1 | Pg | Zn | Zd |

SXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;

### Word

| 31 30 29 28 27 26 25 24 23 22 21 20 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 1 0 1 | Pg | Zn | Zd |

SXTW <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

For the halfword variant: is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);

Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Programmable table lookup in single vector table.

Reads each element of the second source (index) vector and uses its value to select an indexed element from the first source (table) vector, 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.

| 31 30 29 28 27 26 25 24 23 22 21 20 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 | Zm | 0 0 1 1 0 0 | Zn | Zd |

TBL <Zd>.<T>, { <Zn>.<T> }, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
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>B</td>
</tr>
<tr>
<td>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();
n = VL DIV esize;
operand1 = Z[n];
operand2 = Z[m];
result =

for e = 0 to elements-1
  idx = UInt(Elem[operand2, e, esize]);
  Elem[result, e, esize] = if idx < elements then Elem[operand1, idx, esize] else Zeros();

Z[d] = result;
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 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

\[ \text{TRN1 } \langle \text{Pd}\rangle.<\text{T}>, \langle \text{Pn}\rangle.<\text{T}>, \langle \text{Pm}\rangle.<\text{T}> \]

if \( \text{!HaveSVE}() \) then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer n = \text{UInt}(\text{Pn});
integer m = \text{UInt}(\text{Pm});
integer d = \text{UInt}(\text{Pd});
integer part = 0;

### 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>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

\[ \text{TRN2 } \langle \text{Pd}\rangle.<\text{T}>, \langle \text{Pn}\rangle.<\text{T}>, \langle \text{Pm}\rangle.<\text{T}> \]

if \( \text{!HaveSVE}() \) then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer n = \text{UInt}(\text{Pn});
integer m = \text{UInt}(\text{Pm});
integer d = \text{UInt}(\text{Pd});
integer part = 1;

### 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

```plaintext
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;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

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  | Zm | 0  | 1  | 1  | 1  | 0  | 0  | Zn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  

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;

**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  | Zm | 0  | 1  | 1  | 1  | 0  | 0  | Zn | 1 | 1 | 0 | 0 | 1 | Zd |

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;

**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 pairs = \texttt{VL} \ DIV \ (\texttt{esize} \times 2);
bits(\texttt{VL}) \ operand1 = \texttt{Z}[n];
bits(\texttt{VL}) \ operand2 = \texttt{Z}[m];
bits(\texttt{VL}) \ result;

\texttt{for} \ p = 0 \texttt{to} \ pairs - 1
\begin{align*}
\texttt{Elem}[\texttt{result}, 2*p+0, \texttt{esize}] &= \texttt{Elem}[\texttt{operand1}, 2*p+\texttt{part}, \texttt{esize}]; \\
\texttt{Elem}[\texttt{result}, 2*p+1, \texttt{esize}] &= \texttt{Elem}[\texttt{operand2}, 2*p+\texttt{part}, \texttt{esize}];
\end{align*}
\texttt{Z}[d] = \texttt{result};
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.

UABD &lt;Zdn&gt;&lt;T&gt;, &lt;Pg&gt;&lt;/M&gt;, &lt;Zdn&gt;&lt;T&gt;, &lt;Zm&gt;&lt;T&gt;

if !HaveSVE() then UNDEFINED;
integer esize = 8 &lt;&lt; UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler Symbols

&lt;Zdn&gt; Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
&lt;T&gt; Is the size specifier, encoded in "size"

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

&lt;Pg&gt; Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
&lt;Zm&gt; Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
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&lt;esize-1:0&gt;;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UADDV**

Unsigned add reduction to scalar.

Unsigned add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first zero-extended to 64 bits. Inactive elements in the source vector are treated as zero.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | Pg |    | Zn |    | Vd |    |    |    |    |    |    |    |    |    |    |    |    |

UADDV $<$Dd$>$, $<$Pg$>$, $<$Zn$>$.$<$T$>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << $\text{UInt}(\text{size})$;
integer g = $\text{UInt}(\text{Pg})$;
integer n = $\text{UInt}(\text{Zn})$;
integer d = $\text{UInt}(\text{Vd})$;

**Assembler Symbols**

$<$Dd$>$ Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.

$<$Pg$>$ Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

$<$Zn$>$ Is the name of the source scalable vector register, encoded in the "Zn" field.

$<$T$>$ Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;$T$&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

`CheckSVEEnabled();`

integer elements = $\text{VL} \div \text{esize}$;
bits($\text{PL}$) mask = $\text{P}[g]$;
bits($\text{VL}$) operand = $\text{Z}[n]$;
integer sum = 0;

for e = 0 to elements-1
  if $\text{ElemP[mask, e, esize]} == '1'$ then
    integer element = $\text{UInt(Elem[operand, e, esize])}$;
    sum = sum + element;

$\text{V}[d]$ = sum<63:0>;
**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

```
0 1 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 1 0 1 1 0 1 0 1 0 1 0 1 0 1
Pg  Zn  Zd
```

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

```
0 1 1 0 0 1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Pg  Zn  Zd
```

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

```
0 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Pg  Zn  Zd
```

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());
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  |

int_U

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 = 32;
integer d.esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

### 64-bit to half-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |

int_U

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 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  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  |

int_U

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

int_U

UCVTF <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
   bits(esize) element = Elem[operand, e, esize];
   if ElemP[mask, e, esize] == '1'
      bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding);
      Elem[result, e, esize] = ZeroExtend(fpval);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDIV

Unsigned divide (predicated).

Unsigned divide active elements of the first source vector by corresponding elements of the second source vector and
destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the
destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  |
| R  | U  |

UDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer dn = Uint(Zdn);
integer m = Uint(Zm);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>  Is the size specifier, encoded in “size<0>”:

<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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned 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 | 1 | 1 | 1 | 0 | 0 | 0 | Pg | Zm | Zdn |

| R | U |

UDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The **MOVPRFX** instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The **MOVPRFX** instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDOT (vectors)

Unsigned integer dot product.

The unsigned integer dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 0 1 0 0 1 0 0 0 0 1</td>
</tr>
</tbody>
</table>

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>”:

<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 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
  Elem[result, e, esize] = res;
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDOT (indexed)

Unsigned integer indexed dot product.

The unsigned integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```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
```

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 quaduplet 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 quaduplet 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 = \texttt{VL} \texttt{DIV} esize;
integer eltspersegment = 128 \texttt{DIV} 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;

\begin{verbatim}
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = \texttt{Elem}[operand3, e, esize];
    for i = 0 to 3
        integer element1 = \texttt{UInt}(\texttt{Elem}[operand1, 4 \times e + i, esize \texttt{DIV} 4]);
        integer element2 = \texttt{UInt}(\texttt{Elem}[operand2, 4 \times s + i, esize \texttt{DIV} 4]);
        res = res + element1 \times element2;
    \texttt{Elem}[result, e, esize] = res;
\end{verbatim}

\texttt{Z}[da] = result;

Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is UNPREDICTABLE:

\begin{itemize}
    \item The \texttt{MOVPRFX} instruction must be unpredicated.
    \item The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
    \item The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
\end{itemize}
**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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    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**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMAX (immediate)

Unsigned maximum with immediate (unpredicated).

Determine the unsigned maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

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;
biter(VL) operand1 = Z[dn];
biter(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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

\[
\begin{array}{cccccccccccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \mid & Pg & \mid & Zn & \mid & Vd
\end{array}
\]

UMAXV \(<V><d>, <Pg>, <Zn>\):

if !\(\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 << \(\text{UInt}(\text{size})\);
integer g = \(\text{UInt}(\text{Pg})\);
integer n = \(\text{UInt}(\text{Zn})\);
integer d = \(\text{UInt}(\text{Vd})\);
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

\(\text{CheckSVEEnabled}()\);
integer elements = \(\text{VL} \div \text{esize}\);
bits(\(\text{PL}\)) mask = \(\text{P}[g]\);
bits(\(\text{VL}\)) operand = \(\text{Z}[n]\);
integer maximum = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1

\tif \(\text{Elem}[\text{mask, e, esize}] == '1'\) then
\tinteger element = \(\text{Int}(\text{Elem}[\text{operand, e, esize}], \text{unsigned})\);
\tmaximum = \(\text{Max}(\text{maximum, element})\);

\(V[d] = \text{maximum}<\text{esize}-1:0>\);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
</tr>
<tr>
<td>Pg</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>
</tr>
<tr>
<td>Zm</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>
</tr>
<tr>
<td>Zdn</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>
</tr>
</tbody>
</table>
```

UMIN <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":
  - size <T>
    - 00: B
    - 01: H
    - 10: S
    - 11: D
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = 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**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMIN (immediate)**

Unsigned minimum with immediate (unpredicated).

Determine the unsigned minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 1 1 0 | size | 1 0 1 0 1 1 1 0 | imm8 | Zdn
```

**Asmsembler Symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>` Is the unsigned immediate operand, in the range 0 to 255, encoded in the “imm8” field.

**Operation**

```
if !HaveSVE() then UNDEFINED;
integer size = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);

integer elements = VL DIV size;
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**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMINV**

Unsigned minimum reduction to scalar.

Unsigned minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 0 1 0 1 1 0 0 1 Pg Zn Vd</td>
</tr>
</tbody>
</table>

**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**

```plaintext
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>;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
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":
  - `00` B
  - `01` H
  - `10` S
  - `11` D
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQADD (immediate)

Unsigned saturating add immediate (unpredicated).

Unsigned saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to \((2^N)-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);
Z[dn] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQADD (vectors)

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.

```
0 0 0 0 0 1 0 0 | size | 1 | Zm | 0 0 0 1 0 1 | Zn | Zd |
```

UQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “size”:
  - 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;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECB

Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count.

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```
0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 imm4 1 1 1 1 1 1 1{, <pattern>}{, MUL #<imm>}}
```

```java
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

```
0 0 0 0 0 0 1 0 0 0 0 0 1 1 imm4 1 1 1 1 1 1 1{, <pattern>}{, MUL #<imm>}}
```

```java
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";
<pattern>  
| 00000 | POW2   | 00001 | VL1    |
| 00010 | VL2    | 00011 | VL3    |
| 00100 | VL4    | 00101 | VL5    |
| 00110 | VL6    | 00111 | VL7    |
| 01000 | VL8    | 01001 | VL16   |
| 01010 | VL32   | 01011 | VL64   |
| 01100 | VL128  | 01101 | VL256  |
| 01110 | #uimm5 | 101x1 | #uimm5 |
| 10110 | #uimm5 | 1x0x1 | #uimm5 |
| 1x010 | #uimm5 | 1xx00 | #uimm5 |
| 11101 | MUL4   | 11110 | MUL3   |
| 11111 | ALL    |

<imm>  
Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECD (scalar)

Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | IMM4| 1  | 1  | 1  | 1  | 1  | 1  | PATTERN | RDN |

size<1> size<0> sf D U

UQDECD <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | IMM4| 1  | 1  | 1  | 1  | 1  | 1  | PATTERN | RDN |

size<1> size<0> sf D U

UQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern";
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</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>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);  
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECD (vector)

Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 1 1 0</td>
<td>imm4 1 1 0 0 1 1</td>
<td>pattern Zdn</td>
</tr>
</tbody>
</table>

size<1>size<0> D U

UQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQDECH (scalar)

Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|
| 0 0 0 0 0 1 0 0 | 0 1 1 0 | imm4 | 1 1 1 1 1 | pattern | Rdn |

UQDECH <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|
| 0 0 0 0 0 1 0 0 | 0 1 1 1 | imm4 | 1 1 1 1 1 | pattern | Rdn |

UQDECH <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern";
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11110</td>
<td>MUL4</td>
</tr>
<tr>
<td>11111</td>
<td>MUL3</td>
</tr>
<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 count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bias(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
size<1>size<0> D U

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>01100</td>
<td>VL64</td>
</tr>
<tr>
<td>01101</td>
<td>VL128</td>
</tr>
<tr>
<td>01110</td>
<td>VL256</td>
</tr>
<tr>
<td>01111</td>
<td>#uumm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uumm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uumm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uumm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uumm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEenabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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

```
0 0 1 0 0 1 0 | size 1 0 1 0 1 1 1 0 0 1 0 0
D U sf
Pm Rdn
```

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

```
0 0 1 0 0 1 0 1 | size 1 0 1 0 1 1 1 0 0 1 1 0
D U sf
Pm Rdn
```

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

\texttt{CheckSVEEnabled();}
\texttt{integer elements = VL \, DIV \, esize;}
\texttt{bits(ssize) operand1 = X[dn];}
\texttt{bits(PL) operand2 = P[m];}
\texttt{bits(ssize) result;}
\texttt{integer count = 0;}

\texttt{for \, e = 0 \, to \, elements-1}
\texttt{\quad \text{if } \texttt{ElemP}[operand2, \, e, \, esize] \, == \, '1' \, then}
\texttt{\quad \quad count = count + 1;}

\texttt{integer \, element = \texttt{Int}(operand1, \, unsigned);} \quad \texttt{(result, \, -) = \texttt{SatQ}(element \, - \, count, \, ssize, \, unsigned);} \quad \texttt{X[dn] = \texttt{Extend}(result, \, 64, \, unsigned);}
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.

The `Pm` and `Zdn` fields encode the names of the source and destination scalable vector registers, respectively. The `T` field encodes the size specifier, which can be `00` (reserved), `01` (H), `10` (S), or `11` (D). The `unsigned` field determines whether the results should be treated as 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>RESERVED</td>
</tr>
<tr>
<td>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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], '-') = SatQ(element - count, esize, unsigned);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQDECW (scalar)

Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | pattern | Rdn |

size<1> size<0> sf D U

UQDECW <Wdn>, <pattern>{, MUL #<imm>}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
type dn = UInt(Rdn);
bits(5) pat = pattern;
type imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
type 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  | 1  | 0  | 1  | imm4 | 1  | 1  | 1  | 1  | 1  | pattern | Rdn |

size<1> size<0> sf D U

UQDECW <Xdn>, <pattern>{, MUL #<imm>}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
type dn = UInt(Rdn);
bits(5) pat = pattern;
type imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
type 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>10111</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);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECW (vector)

Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23</th>
<th>22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size&lt;1&gt;size&lt;0&gt;</th>
<th>D U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 0 1 1 0 0 1 1</td>
<td>pattern</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>

UQDECW <Zdn>.S{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</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>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>MUL4</td>
</tr>
<tr>
<td>11100</td>
<td>MUL3</td>
</tr>
<tr>
<td>11110</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
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  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | pattern | Rdn |
| size<1>|size<0> | sf | D | U |

UQINCB <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | pattern | Rdn |
| size<1>|size<0> | sf | D | U |

UQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
<table>
<thead>
<tr>
<th>Pattern</th>
<th>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>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</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];
bites(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQINCD (scalar)**

Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

![32-bit encoding](image)

```javascript
UQINCD <Wdn>{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

### 64-bit

![64-bit encoding](image)

```javascript
UQINCD <Xdn>{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

**Assembler Symbols**

- `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01100</td>
<td>VL64</td>
</tr>
<tr>
<td>01101</td>
<td>VL128</td>
</tr>
<tr>
<td>01110</td>
<td>VL256</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCD (vector)

Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>Zdn</th>
<th>pattern</th>
<th>imm4</th>
<th>size&lt;0&gt;</th>
<th>size&lt;1&gt;</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

UQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern> is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

| 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>01010</td>
<td>VL16</td>
</tr>
<tr>
<td>01011</td>
<td>VL32</td>
</tr>
<tr>
<td>01111</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], '-') = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQINCH (scalar)

Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
</tr>
</tbody>
</table>

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

UQINCH (scalar)
<pattern> | <pattern>
---|---
00000 | POW2
00001 | VL1
00010 | VL2
00011 | VL3
00100 | VL4
00101 | VL5
00110 | VL6
00111 | VL7
01000 | VL8
01001 | VL16
01010 | VL32
01011 | VL64
01100 | VL128
01101 | VL256
0111x | #uimm5
101x1 | #uimm5
10110 | #uimm5
1x0x1 | #uimm5
1x010 | #uimm5
1xx00 | #uimm5
11101 | MUL4
11110 | MUL3
11111 | ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCH (vector)

Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<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>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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>D</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQINCP (scalar)

Unsigned saturating increment scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | size | 1 0 1 0 0 | Pm | Rdn
```

UQINCP <Wdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;

### 64-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | size | 1 0 1 0 0 | Pm | Rdn
```

UQINCP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;

### Assembler Symbols

- `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- `<Pm>` Is the name of the source scalable predicate register, encoded in the "Pm" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;
end for

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
UQINCP (vector)

Unsigned saturating increment vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | U  |    |    |    | size | 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  |

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQINC (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  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | pattern |

UQINC6 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | pattern |

UQINC6 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>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11001</td>
<td>MUL4</td>
</tr>
<tr>
<td>11100</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
<table>
<thead>
<tr>
<th>field</th>
<th>encoding</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm4</td>
<td>0-15</td>
<td>immediate</td>
</tr>
<tr>
<td>Zdn</td>
<td>0-255</td>
<td>predicate</td>
</tr>
<tr>
<td>pattern</td>
<td>0-15</td>
<td>pattern</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>direction</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>direction</td>
</tr>
</tbody>
</table>
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

```
<table>
<thead>
<tr>
<th>pattern</th>
<th>encoding</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>VL16</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>VL32</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>VL64</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>VL128</td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td>VL256</td>
<td></td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
<td></td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
<td></td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
<td></td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
<td></td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td>ALL</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.

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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".

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

\(\text{CheckSVEEnabled}()\);
integer elements = \(\text{VL} \div \text{esize}\);
bits(\text{VL}) \text{ operand1} = Z[\text{dn}];
bits(\text{VL}) \text{ result};
for \(e = 0\) to elements-1
    integer element1 = \(\text{Int}(\text{Elem} [\text{operand1}, e, \text{esize}], \text{unsigned})\);
    (\text{Elem}[\text{result}, e, \text{esize}, \cdot]) = \text{SatQ}(\text{element1} - \text{imm}, \text{esize}, \text{unsigned});
\]
\(Z[\text{dn}] = \text{result};\)

Operational information

This instruction might be immediately preceded in program order by a \text{MOVPRFX} instruction. The \text{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \text{MOVPRFX} and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned saturating 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 0 1 0 0 | size | 1 | Zm | 0 0 0 1 1 | 1 | Zn | Zd |
```

UQSUB \(<Zd>\), \(<T>\), \(<Zn>\), \(<T>\), \(<Zm>\), \(<T>\)

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;

### Assembler Symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);
Z[d] = result;
UUNPKHI, UUNPKLO

Unsigned unpack and extend half of vector.

Unpack elements from the lowest or highest half of the source vector and then zero-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: High half and Low half

**High half**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|---|
| 0 0 0 0 0 0 1 0 1 | size | 1 1 0 0 1 1 0 1 1 1 0 | Zn | Zd |
```

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**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|---|
| 0 0 0 0 0 0 1 0 1 | size | 1 1 0 0 1 0 0 1 1 1 0 | Zn | Zd |
```

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

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;
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**

| 31 30 29 28 27 26 25 24 23 22 21 20 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 1 0 0 | Pg  | Zn  | Zd  |

UXTB <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 = TRUE;

**Halfword**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0 | 0 1 0 0 1 1 1 0 1 | Pg  | Zn  | Zd  |

UXTH <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 = TRUE;

**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 0 | 0 1 0 1 0 1 1 0 1 | Pg  | Zn  | Zd  |

UXTW <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

For the halfword variant: is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
   bits(esize) element = Elem[operand, e, esize];
   if Elem[mask, e, esize] == '1' then
      Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{H} \]

UZP1 \(<Pd>.<T>, <Pn>.<T>, <Pm>.<T>\)

if ![HaveSVE]() then UNDEFINED;
integer esize = 8 << \text{lnt}(size);
integer n = \text{lnt}(Pn);
integer m = \text{lnt}(Pm);
integer d = \text{lnt}(Pd);
integer part = 0;

### 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>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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{H} \]

UZP2 \(<Pd>.<T>, <Pn>.<T>, <Pm>.<T>\)

if ![HaveSVE]() then UNDEFINED;
integer esize = 8 << \text{lnt}(size);
integer n = \text{lnt}(Pn);
integer m = \text{lnt}(Pm);
integer d = \text{lnt}(Pd);
integer part = 1;

**Assembler Symbols**

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pn>** Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- **<Pm>** Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} esize;

\texttt{bits(PL)} operand1 = \texttt{P}[n];
\texttt{bits(PL)} operand2 = \texttt{P}[m];

\texttt{bits(PL)} result;

\texttt{bits(PL+2)} zipped = operand2:operand1;

\texttt{for} e = 0 \texttt{to} elements-1
\texttt{Elem[result, e, esize DIV 8]} = \texttt{Elem[zipped, 2*e+part, esize DIV 8]};

\texttt{P[d]} = result;
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.

It has encodings from 2 classes: Even and Odd

Even

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
|0 |0 |0 |0 |0 |0 |1 |0 |1 |0 |0 |0 |0 |0 |1 |0 |1 |0 |0 |0 |0 |0 |0 |1 |0 |1 |0 |0 |0 |0 |1 |0 |0 |0 |0 |

H

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;

Odd

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
|0 |0 |0 |0 |0 |0 |1 |0 |1 |0 |1 |0 |1 |0 |0 |0 |0 |1 |0 |1 |0 |0 |1 |0 |1 |0 |0 |0 |0 |1 |0 |0 |0 |0 |

H

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;

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;

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;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
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;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
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

<table>
<thead>
<tr>
<th>&lt;Pd&gt;</th>
<th>Is the name of the destination scalable predicate register, encoded in the &quot;Pd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;T&gt;</td>
<td>Is the size specifier, encoded in “size”:</td>
</tr>
<tr>
<td></td>
<td>size   &lt;T&gt;</td>
</tr>
<tr>
<td></td>
<td>00     B</td>
</tr>
<tr>
<td></td>
<td>01     H</td>
</tr>
<tr>
<td></td>
<td>10     S</td>
</tr>
<tr>
<td></td>
<td>11     D</td>
</tr>
<tr>
<td>&lt;R&gt;</td>
<td>Is a width specifier, encoded in “sf”:</td>
</tr>
<tr>
<td></td>
<td>sf    &lt;R&gt;</td>
</tr>
<tr>
<td></td>
<td>0     W</td>
</tr>
<tr>
<td></td>
<td>1     X</td>
</tr>
<tr>
<td>&lt;n&gt;</td>
<td>Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.</td>
</tr>
<tr>
<td>&lt;m&gt;</td>
<td>Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.</td>
</tr>
</tbody>
</table>
Operation

\texttt{CheckSVEEnabled}();
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(PL) mask = \texttt{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 \texttt{Cmp\_LT} cond = (\texttt{Int}(operand1, unsigned) < \texttt{Int}(operand2, unsigned));
    when \texttt{Cmp\_LE} cond = (\texttt{Int}(operand1, unsigned) <= \texttt{Int}(operand2, unsigned));

  last = last && cond;
  \texttt{ElemP}[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 + 1;

\texttt{PSTATE.<N,Z,C,V>} = \texttt{PredTest}(mask, result, esize);
\texttt{P[d]} = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 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 |

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

```
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;
```
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.

```
0 0 1 0 0 1 0 1 | size | 1 | Rm | 0 0 0 | sf | 0 1 | Rn | 0 | Pd
```

```
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

CheckSVEEnabled();
integer elements = \text{VL} \text{DIV} \text{esize};
bits(PL) mask = \text{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 = (\text{Int}(operand1, unsigned) < \text{Int}(operand2, unsigned));
    when Cmp_LE cond = (\text{Int}(operand1, unsigned) <= \text{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;
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 0 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0

WRFFR <Pn>.B
```

```
if !HaveSVE() then UNDEFINED;
integer n = UInt(Pn);
```

Assemble Symbol

- `<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;
```
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</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>
</tbody>
</table>

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</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>
</tbody>
</table>

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;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bts(PL) operand1 = P[n];
bts(PL) operand2 = P[m];
bts(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;
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.

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>Zm</th>
<th>0 1 1 0 0 1</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

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;

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>Zm</th>
<th>0 1 1 0 0 1</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

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;

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 pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) 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];`

Z[d] = result;
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.

### SVE encodings

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>0000000000</td>
<td>UDF</td>
</tr>
<tr>
<td>!= 000</td>
<td>0000000000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>!= 000</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x</td>
<td>0xxxx</td>
<td>SVE Integer Multiply-Add - Predicated</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>0xxxx</td>
<td>SVE Integer Binary Arithmetic - Predicated</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>001xxxx</td>
<td>SVE Integer Reduction</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>0xxxx</td>
<td>SVE Bitwise Shift - Predicated</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>101xxxx</td>
<td>SVE Integer Unary Arithmetic - Predicated</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>1xxxx</td>
<td>SVE integer add/subtract vectors (unpredicated)</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>1xxxx</td>
<td>SVE Bitwise Logical - Unpredicated</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>1xxxx</td>
<td>SVE Index Generation</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>1xxxx</td>
<td>SVE Stack Allocation</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>011xxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>1xxxx</td>
<td>SVE Bitwise Shift - Unpredicated</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>1xxxx</td>
<td>SVE address generation</td>
</tr>
<tr>
<td>0000</td>
<td>0x</td>
<td>1xxxx</td>
<td>SVE Integer Misc - Unpredicated</td>
</tr>
<tr>
<td></td>
<td>0x</td>
<td>1xxxx</td>
<td>11xxxx</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>11xxxx</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>00xxx</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>01xxx</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>001000</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>001001</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>00101x</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>0011x1</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>001100</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>001110</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>010xxx</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>011xxx</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>10xxx</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxx</td>
<td>11xxx</td>
</tr>
<tr>
<td>000</td>
<td>10</td>
<td>1xxxx</td>
<td>000xx</td>
</tr>
<tr>
<td>000</td>
<td>11</td>
<td>1xxxx</td>
<td>000xx</td>
</tr>
<tr>
<td>001</td>
<td>0x</td>
<td>0xxxx</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0x</td>
<td>1xxxx</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>0xxxx</td>
<td>x0xxxx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>0xxxx</td>
<td>01xxxx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>0xxxx</td>
<td>11xxxx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>01xxx</td>
<td>01xxxx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>01xxx</td>
<td>11xxxx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxxx</td>
<td>000xx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxxx</td>
<td>01xxx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxxx</td>
<td>11xxx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxxx</td>
<td>100xx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx</td>
<td>1000xx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx</td>
<td>1001xx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx</td>
<td>101xxx</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>11xxx</td>
<td>10xxx</td>
</tr>
<tr>
<td>010</td>
<td>0x</td>
<td>0xxxx</td>
<td>0xxxxx</td>
</tr>
<tr>
<td>010</td>
<td>0x</td>
<td>0xxxx</td>
<td>1xxxxx</td>
</tr>
<tr>
<td>010</td>
<td>0x</td>
<td>1xxxx</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>0xxxx</td>
<td>0xxxxx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>00000</td>
<td>100xx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>00000</td>
<td>101xxx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>00000</td>
<td>11xxx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>00001</td>
<td>1xxxxx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>0001x</td>
<td>1xxxxx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>001xx</td>
<td>1xxxxx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>01xxx</td>
<td>1xxxxx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>1xxxx</td>
<td>00x01x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>1xxxx</td>
<td>00000x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>1xxxx</td>
<td>0001xx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>1xxxx</td>
<td>001000</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>1xxxx</td>
<td>001001</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>1xxxx</td>
<td>0011xx</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>1xxxx</td>
<td>01xxxx</td>
</tr>
</tbody>
</table>
SVE Integer Multiply-Add - Predicated

These instructions are under SVE encodings.

```plaintext
00000100 1 0
```

<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.

```plaintext
0 0 0 0 0 1 0 0 size 0 Zm 0 1 op Pg Zn Zda
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MLA</td>
</tr>
<tr>
<td>1</td>
<td>MLS</td>
</tr>
</tbody>
</table>

SVE integer multiply-add writing multiplicand (predicated)

These instructions are under SVE Integer Multiply-Add - Predicated.

```plaintext
0 0 0 0 0 1 0 0 size 0 Zm 1 1 op Pg Za Zdn
```
SVE Integer Binary Arithmetic - Predicated

These instructions are under SVE encodings.

### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MAD</td>
</tr>
<tr>
<td>1</td>
<td>MSB</td>
</tr>
</tbody>
</table>

### SVE Integer Binary Arithmetic - Predicated

These instructions are under SVE Integer Binary Arithmetic - Predicated.

#### SVE integer add/subtract vectors (predicated)

These instructions are under SVE Integer Binary Arithmetic - Predicated.

#### SVE integer min/max/difference (predicated)

These instructions are under SVE Integer Binary Arithmetic - Predicated.

#### SVE bitwise logical operations (predicated)
### SVE integer multiply vectors (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

<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)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>SMULH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>UMULH</td>
</tr>
</tbody>
</table>

### SVE integer divide vectors (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

<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](#).

<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>opc</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE integer add reduction (predicated)</td>
</tr>
<tr>
<td>01</td>
<td>SVE integer min/max reduction (predicated)</td>
</tr>
<tr>
<td>10</td>
<td>SVE constructive prefix (predicated)</td>
</tr>
<tr>
<td>11</td>
<td>SVE bitwise logical reduction (predicated)</td>
</tr>
</tbody>
</table>
SVE integer add reduction (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>SADDV</td>
</tr>
<tr>
<td>00 1</td>
<td>UADDV</td>
</tr>
<tr>
<td>01 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE integer min/max reduction (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>SMAXV</td>
</tr>
<tr>
<td>00 1</td>
<td>UMAXV</td>
</tr>
<tr>
<td>01 0</td>
<td>SMINV</td>
</tr>
<tr>
<td>01 1</td>
<td>UMINV</td>
</tr>
<tr>
<td>1x 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE constructive prefix (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>MOVPRFX (predicated)</td>
</tr>
<tr>
<td>01 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE bitwise logical reduction (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0</td>
<td>ORV</td>
</tr>
<tr>
<td>001 0</td>
<td>EORV</td>
</tr>
<tr>
<td>010 0</td>
<td>ANDV</td>
</tr>
<tr>
<td>011 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE Bitwise Shift - Predicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0 0 1 0 0</td>
<td>ASR (immediate, predicated)</td>
</tr>
<tr>
<td>00 0 1 1</td>
<td>LSR (immediate, predicated)</td>
</tr>
<tr>
<td>00 1 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 1 1 1</td>
<td>LSL (immediate, predicated)</td>
</tr>
<tr>
<td>01 0 0 1</td>
<td>ASRD</td>
</tr>
<tr>
<td>01 0 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 1 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE bitwise shift by vector (predicated)

These instructions are under SVE Bitwise Shift - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 1 0 0</td>
<td>ASR (vectors)</td>
</tr>
<tr>
<td>0 0 1 1 1</td>
<td>LSR (vectors)</td>
</tr>
<tr>
<td>0 1 1 1 1</td>
<td>LSL (vectors)</td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>ASRR</td>
</tr>
<tr>
<td>1 0 1 1 1</td>
<td>LSRR</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>LSLR</td>
</tr>
</tbody>
</table>

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](#).

<table>
<thead>
<tr>
<th>R</th>
<th>L</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ASR (wide elements, predicated)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>LSR (wide elements, predicated)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>LSL (wide elements, predicated)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE Integer Unary operations (predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>R</th>
<th>L</th>
<th>U</th>
<th>size</th>
<th>opc</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<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>SXTB, SXTH, SXTW — SXTB</td>
</tr>
<tr>
<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>UXTB, UXTH, UXTW — UXTB</td>
</tr>
<tr>
<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>SXTB, SXTH, SXTW — SXTH</td>
</tr>
<tr>
<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>UXTB, UXTH, UXTW — UXTH</td>
</tr>
<tr>
<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>SXTB, SXTH, SXTW — SXTW</td>
</tr>
<tr>
<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>UXTB, UXTH, UXTW — UXTW</td>
</tr>
<tr>
<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>ABS</td>
</tr>
<tr>
<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>NEG</td>
</tr>
</tbody>
</table>

### SVE bitwise unary operations (predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>R</th>
<th>L</th>
<th>U</th>
<th>size</th>
<th>opc</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<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>CLS</td>
</tr>
<tr>
<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>CNT</td>
</tr>
<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>CNOP</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FABS</td>
</tr>
<tr>
<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>FNEG</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td><strong>NOT (vector)</strong></td>
</tr>
<tr>
<td>111</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
</tbody>
</table>

### SVE integer add/subtract vectors (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>0 0 0 0 0 1 0 0 size 1</td>
</tr>
<tr>
<td>Zm 0 0 0 opc Zn Zd</td>
</tr>
</tbody>
</table>

### SVE Bitwise Logical - 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>00000100 1</td>
</tr>
<tr>
<td>001 op0 op1</td>
</tr>
</tbody>
</table>

### SVE bitwise logical operations (unpredicated)

These instructions are under [SVE Bitwise Logical - 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 opc 1</td>
</tr>
<tr>
<td>Zm 0 0 1 1 0 0 Zn Zd</td>
</tr>
</tbody>
</table>

### SVE Index Generation

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>00000100 1</td>
</tr>
<tr>
<td>0100 op0</td>
</tr>
</tbody>
</table>

---

*SVE Index Generation* is a set of instructions that manipulate index registers. These instructions are under [SVE encodings](#).
SVE Stack Allocation

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>00000100 op0 1 0101 op1</td>
</tr>
</tbody>
</table>
```

SVE stack frame adjustment

These instructions are under SVE Stack Allocation.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 op 1 Rn 0 1 0 1 0 imm6 Rd</td>
</tr>
</tbody>
</table>
```

SVE stack frame size

These instructions are under SVE Stack Allocation.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 op 1 opc2 0 1 0 1 0 imm6 Rd</td>
</tr>
</tbody>
</table>
```

SVE Bitwise Shift - 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>00000100 1 100 op0</td>
</tr>
</tbody>
</table>
```
### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE bitwise shift by wide elements (unpredicated)</td>
</tr>
<tr>
<td>1</td>
<td>SVE bitwise shift by immediate (unpredicated)</td>
</tr>
</tbody>
</table>

### SVE bitwise shift by wide elements (unpredicated)

These instructions are under [SVE Bitwise Shift - Unpredicated](#).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 1 0 0</th>
<th>size</th>
<th>Zm</th>
<th>1 0 0 0</th>
<th>opc</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ASR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>LSR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LSL (wide elements, unpredicated)</td>
</tr>
</tbody>
</table>

### SVE bitwise shift by immediate (unpredicated)

These instructions are under [SVE Bitwise Shift - Unpredicated](#).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 1 0 0</th>
<th>tszh</th>
<th>tszl</th>
<th>imm3</th>
<th>1 0 0 1</th>
<th>opc</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

### SVE address generation

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>0 0 0 0 1 0 0</th>
<th>opc</th>
<th>1</th>
<th>Zm</th>
<th>1 0 1 0</th>
<th>msz</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ASR (immediate, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>LSR (immediate, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LSL (immediate, unpredicated)</td>
</tr>
</tbody>
</table>

### SVE Integer Misc - 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>
<th>00000100</th>
<th>1</th>
<th>1011</th>
<th>op0</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADR — Unpacked 32-bit signed offsets</td>
</tr>
<tr>
<td>01</td>
<td>ADR — Unpacked 32-bit unsigned offsets</td>
</tr>
<tr>
<td>1x</td>
<td>ADR — Packed offsets</td>
</tr>
<tr>
<td>0x</td>
<td>SVE floating-point trig select coefficient</td>
</tr>
<tr>
<td>----</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>10</td>
<td>SVE floating-point exponential accelerator</td>
</tr>
<tr>
<td>11</td>
<td>SVE constructive prefix (unpredicated)</td>
</tr>
</tbody>
</table>

**SVE floating-point trig select coefficient**

These instructions are under [SVE Integer Misc - Unpredicated](#).

![Decode fields](image)

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FTSSEL</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE floating-point exponential accelerator**

These instructions are under [SVE Integer Misc - Unpredicated](#).

![Decode fields](image)

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>FEXPA</td>
</tr>
<tr>
<td>00001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xxxx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE constructive prefix (unpredicated)**

These instructions are under [SVE Integer Misc - Unpredicated](#).

![Decode fields](image)

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>MOVPRFX (unpredicated)</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE Element Count**

These instructions are under [SVE encodings](#).
### SVE saturating inc/dec vector by element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 0 00x</td>
<td>SVE saturating inc/dec vector by element count</td>
</tr>
<tr>
<td>op0 1 100</td>
<td>SVE element count</td>
</tr>
<tr>
<td>op1 0 101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op1 1 000</td>
<td>SVE inc/dec vector by element count</td>
</tr>
<tr>
<td>op1 1 100</td>
<td>SVE inc/dec register by element count</td>
</tr>
<tr>
<td>op1 x01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op1 01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op1 11x</td>
<td>SVE saturating inc/dec register by element count</td>
</tr>
</tbody>
</table>

### SVE element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size D U</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 0 0</td>
<td>SQINCH (vector)</td>
</tr>
<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</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) — INCW</td>
</tr>
<tr>
<td>11 1</td>
<td>DECD, DECH, DECW (vector) — DECH</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>INCB, INCD, INCH, INCW (scalar) — INCB</td>
</tr>
<tr>
<td>00 1</td>
<td>DECB, 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>DECB, DECD, DECH, DECW (scalar) — DECH</td>
</tr>
<tr>
<td>10 0</td>
<td>INCB, INCD, INCH, INCW (scalar) — INCH</td>
</tr>
<tr>
<td>10 1</td>
<td>DECB, DECD, DECH, DECW (scalar) — DECH</td>
</tr>
<tr>
<td>11 0</td>
<td>INCB, INCD, INCH, INCW (scalar) — INCW</td>
</tr>
<tr>
<td>11 1</td>
<td>DECB, 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 0</td>
<td>SQINCB — 32-bit</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td>UQINCB — 32-bit</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td>SQDECB — 32-bit</td>
</tr>
<tr>
<td>00 0 1 1</td>
<td>UQDECB — 32-bit</td>
</tr>
<tr>
<td>00 1 0 0</td>
<td>SQINCB — 64-bit</td>
</tr>
<tr>
<td>00 1 0 1</td>
<td>UQINCB — 64-bit</td>
</tr>
<tr>
<td>00 1 1 0</td>
<td>SQDECB — 64-bit</td>
</tr>
<tr>
<td>00 1 1 1</td>
<td>UQDECB — 64-bit</td>
</tr>
<tr>
<td>01 0 0 0</td>
<td>SQINCH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 0 0 1</td>
<td>UQINCH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 0 1 0</td>
<td>SQDECH (scalar) — 32-bit</td>
</tr>
</tbody>
</table>
### SVE Bitwise Immediate

These instructions are under SVE encodings.

![Instruction Details Table]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 0 1 1</td>
<td>UQDECH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 1 0 0</td>
<td>SQINCH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 0 1</td>
<td>UQINCH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 1 0</td>
<td>SQDECH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 1 1</td>
<td>UQDECH (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 0 0 0</td>
<td>SQINCW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 0 1 0</td>
<td>SQDECW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 0 1 1</td>
<td>UQDECW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 1 0 0</td>
<td>SQINCW (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 1 0 1</td>
<td>SQINCW (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 1 1 0</td>
<td>SQDECW (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 1 1 1</td>
<td>UQDECW (scalar) — 64-bit</td>
</tr>
<tr>
<td>11 0 0 0</td>
<td>SQINCD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 0 1</td>
<td>UQINCD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 1 0</td>
<td>SQDECD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 1 1</td>
<td>UQDECD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 1 0 0</td>
<td>SQINCD (scalar) — 64-bit</td>
</tr>
<tr>
<td>11 1 0 1</td>
<td>UQINCD (scalar) — 64-bit</td>
</tr>
<tr>
<td>11 1 1 0</td>
<td>SQDECD (scalar) — 64-bit</td>
</tr>
<tr>
<td>11 1 1 1</td>
<td>UQDECD (scalar) — 64-bit</td>
</tr>
</tbody>
</table>

![Decode fields]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 00</td>
<td>DUPM</td>
</tr>
<tr>
<td>!= 11 00</td>
<td>SVE bitwise logical with immediate (unpredicated)</td>
</tr>
<tr>
<td>!= 00</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE bitwise logical with immediate (unpredicated)

These instructions are under SVE Bitwise Immediate.

![Instruction Details Table]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1</td>
<td>ORR (immediate)</td>
</tr>
<tr>
<td>!= 11 0 0 0 0</td>
<td>imm13</td>
</tr>
<tr>
<td>Zdn</td>
<td>opc</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 11 & opc != 11

![Decode fields]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ORR (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>EOR (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>AND (immediate)</td>
</tr>
</tbody>
</table>
SVE Integer Wide Immediate - Predicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xX</td>
<td>SVE copy integer immediate (predicated)</td>
</tr>
<tr>
<td>10x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>FCPY</td>
</tr>
<tr>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE copy integer immediate (predicated)

These instructions are under SVE Integer Wide Immediate - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</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>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>DUP (scalar)</td>
</tr>
<tr>
<td>00 100</td>
<td>INSR (scalar)</td>
</tr>
<tr>
<td>00 x10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 xx1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 0xX</td>
<td>SVE unpack vector elements</td>
</tr>
<tr>
<td>10 100</td>
<td>INSR (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>10 110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 1x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 000</td>
<td>REV (vector)</td>
</tr>
<tr>
<td>11 != 000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE unpack vector elements

These instructions are under SVE Permute Vector - Unpredicated.
SVE Permute Predicate

These instructions are under SVE encodings.

### SVE unpack predicate elements

These instructions are under SVE Permute Predicate.

### SVE permute predicate elements

These instructions are under SVE Permute Predicate.
### SVE permute vector elements

These instructions are under [SVE encodings](#).

```
+-----------------+----------------+---------------+
| opc  H          | Instruction Details                      |
|-----------------+----------------+---------------+
| 00              | ZIP1, ZIP2 (predicates) — ZIP2            |
| 01 0            | UZP1, UZP2 (predicates) — UZP1           |
| 01 1            | UZP1, UZP2 (predicates) — UZP2           |
| 10 0            | TRN1, TRN2 (predicates) — TRN1           |
| 10 1            | TRN1, TRN2 (predicates) — TRN2           |
| 11              | UNALLOCATED                                  |
```

### SVE Permute Vector - Predicated

These instructions are under [SVE encodings](#).

```
+-----------------+----------------+---------------+
| opc  H          | Instruction Details                      |
|-----------------+----------------+---------------+
| 000             | ZIP1, ZIP2 (vectors) — ZIP1               |
| 001             | ZIP1, ZIP2 (vectors) — ZIP2               |
| 010             | UZP1, UZP2 (vectors) — UZP1              |
| 011             | UZP1, UZP2 (vectors) — UZP2              |
| 100             | TRN1, TRN2 (vectors) — TRN1              |
| 101             | TRN1, TRN2 (vectors) — TRN2              |
| 11x             | UNALLOCATED                                  |
```

![Top-level encodings for A64](image-url)
SVE extract element to general register

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>1</th>
<th>000</th>
<th>0</th>
<th>UNALLOCATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>000</td>
<td>1</td>
<td>SVE conditionally extract element to general register</td>
</tr>
<tr>
<td>1</td>
<td>!= 000</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LASTA (scalar)</td>
</tr>
<tr>
<td>1</td>
<td>LASTB (scalar)</td>
</tr>
</tbody>
</table>

SVE extract element to SIMD&FP scalar register

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LASTA (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>1</td>
<td>LASTB (SIMD&amp;FP scalar)</td>
</tr>
</tbody>
</table>

SVE reverse within elements

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>REVB, REVH, REVW — REVB</td>
</tr>
<tr>
<td>01</td>
<td>REVB, REVH, REVW — REVH</td>
</tr>
<tr>
<td>10</td>
<td>REVB, REVH, REVW — REVW</td>
</tr>
<tr>
<td>11</td>
<td>RBIT</td>
</tr>
</tbody>
</table>

SVE conditionally broadcast element to vector

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>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**.

<table>
<thead>
<tr>
<th>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**.

<table>
<thead>
<tr>
<th>B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLASTA (scalar)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (scalar)</td>
</tr>
</tbody>
</table>

### SVE Permute Vector - Extract

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EXT</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE Integer Compare - Vectors

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE integer compare vectors</td>
</tr>
<tr>
<td>1</td>
<td>SVE integer compare with wide elements</td>
</tr>
</tbody>
</table>

### SVE integer compare vectors

These instructions are under **SVE Integer Compare - Vectors**.
### Decode fields op o2 ne Instruction Details

<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) — CMPHS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (vectors) — CMPHI</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) — CMPNE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPGE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (vectors) — CMPGT</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPEQ</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (vectors) — CMPNE</td>
</tr>
</tbody>
</table>

### SVE integer compare with wide elements

These instructions are under [SVE Integer Compare - Vectors](#).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 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  | size | 0  | Zm | U | 1 | lt | Pg | Zn | ne | Pd |

### Decode fields U lt ne Instruction Details

<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) — CMPGE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPGT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLT</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPEQ</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPHI</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPSL</td>
</tr>
</tbody>
</table>

### SVE integer compare with unsigned immediate

These instructions are under [SVE encodings](#).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | size | 1  | imm7 | lt | Pg | Zn | ne | Pd |

### Decode fields lt ne Instruction Details

<table>
<thead>
<tr>
<th>lt</th>
<th>ne</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPHS</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPHI</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPLQ</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPLS</td>
</tr>
</tbody>
</table>

### SVE integer compare with signed immediate

These instructions are under [SVE encodings](#).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | size | 0  | imm5 | op | 0 | o2 | Pg | Zn | ne | Pd |

### Decode fields op o2 ne Instruction Details

<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; (immediate) — CMPGE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPGT</td>
</tr>
</tbody>
</table>

---

Top-level encodings for A64
### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>o2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPLT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPEQ</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPEQ</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate logical operations

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>SVE Predicate Logical Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100101</td>
</tr>
</tbody>
</table>

### SVE Propagate Break

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100101</td>
<td>SVE propagate break from previous partition</td>
</tr>
<tr>
<td>00101</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE propagate break from previous partition

These instructions are under SVE Propagate Break.

<table>
<thead>
<tr>
<th>0 0100101</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE propagate break from previous partition</td>
<td></td>
</tr>
<tr>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
### SVE Partition Break

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BRKPA, BRKPAS — not setting the condition flags</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>BRKPB, BRKPBS — not setting the condition flags</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>BRKPA, BRKPAS — setting the condition flags</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>BRKPB, BRKPBS — setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### SVE propagate break to next partition

These instructions are under **SVE Partition Break**.

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode fields</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SVE propagate break to next partition</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>0</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x000</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xx1x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xxx1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>0</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0</td>
<td></td>
<td></td>
<td>SVE partition break condition</td>
</tr>
</tbody>
</table>

#### SVE partition break condition

These instructions are under **SVE Partition Break**.

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode fields</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>B</td>
<td>BRKA, BRKAS — not setting the condition flags</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>BRKA, BRKAS — setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>BRKB, BRKBS — not setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>BRKB, BRKBS — setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

*Page 2334*
SVE Predicate Misc

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x0</td>
<td>x0</td>
<td>0</td>
<td>SVE predicate test</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>x0</td>
<td>x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>x0</td>
<td>x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0xx1</td>
<td>x0</td>
<td>x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0xxx</td>
<td>x1</td>
<td>x1</td>
<td>0</td>
<td>UNALLOCATED</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 !0</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 10</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>
<tr>
<td>1001 100 10</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 110 10 0000</td>
<td>0</td>
<td>SVE predicate read from FFR (unpredicated)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 110 10 !0000</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100x 010</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100x 100 0x</td>
<td>0</td>
<td>SVE predicate initialize</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100x 100 11</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100x 110 10</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100x xx1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110x</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x1x</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</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 S opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 01xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</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>

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>PFALSE</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>

**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></td>
<td>RDFFR, RDFFRS (predicated) — not setting the condition flags</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RDFFR, RDFFRS (predicated) — setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE predicate read from FFR (predicated)**

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>RDFFR (unpredicated)</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>

**SVE predicate read from FFR (unpredicated)**

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>RDFFR (unpredicated)</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>
### SVE Integer Compare - Scalars

These instructions are under **SVE encodings**.

#### SVE integer compare scalar count and limit

These instructions are under **SVE Integer Compare - Scalars**.

#### SVE conditionally terminate scalars

These instructions are under **SVE Integer Compare - Scalars**.

#### SVE Integer Wide Immediate - Unpredicated

These instructions are under **SVE encodings**.
### SVE integer add/subtract immediate (unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>SVE integer add/subtract immediate (unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>SVE integer min/max immediate (unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>SVE integer multiply immediate (unpredicated)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>SVE broadcast integer immediate (unpredicated)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>SVE broadcast floating-point immediate (unpredicated)</td>
</tr>
</tbody>
</table>

### SVE integer min/max 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>opc</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>ADD (immediate)</td>
</tr>
<tr>
<td>001</td>
<td>SUB (immediate)</td>
</tr>
<tr>
<td>010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>SUBR (immediate)</td>
</tr>
<tr>
<td>100</td>
<td>SQADD (immediate)</td>
</tr>
<tr>
<td>101</td>
<td>UQADD (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>S QSUB (immediate)</td>
</tr>
<tr>
<td>111</td>
<td>UQSUB (immediate)</td>
</tr>
</tbody>
</table>

### SVE integer multiply 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>opc o2</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>MUL (immediate)</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</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>opc o2</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>sh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zd</td>
<td></td>
<td></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>opc o2</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>DUP (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>o2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zd</td>
<td></td>
<td></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>opc o2</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FDUP</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Pg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></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>opc o2</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100101</td>
<td>101</td>
<td>op0</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>o2</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>CNTP</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</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 D</th>
<th>U</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 00</td>
<td></td>
<td></td>
<td>SQINCP (vector)</td>
</tr>
<tr>
<td>0 1 00</td>
<td></td>
<td></td>
<td>UQINCP (vector)</td>
</tr>
<tr>
<td>1 0 00</td>
<td></td>
<td></td>
<td>SQDECP (vector)</td>
</tr>
<tr>
<td>1 1 00</td>
<td></td>
<td></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 D</th>
<th>U</th>
<th>sf</th>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td></td>
<td>32-bit</td>
<td>SQINCP (scalar)</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td></td>
<td>64-bit</td>
<td>SQINCP (scalar)</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
<td>32-bit</td>
<td>UQINCP (scalar)</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td></td>
<td>64-bit</td>
<td>UQINCP (scalar)</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td></td>
<td>32-bit</td>
<td>SQDECP (scalar)</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
<td>64-bit</td>
<td>SQDECP (scalar)</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td></td>
<td>32-bit</td>
<td>UQDECP (scalar)</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
<td>64-bit</td>
<td>UQDECP (scalar)</td>
<td></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 D</th>
<th>opc</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 01</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 00</td>
<td></td>
<td></td>
<td>INCP (vector)</td>
</tr>
<tr>
<td>0 1 00</td>
<td></td>
<td></td>
<td>DECP (vector)</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>size</th>
<th>10</th>
<th>01</th>
<th>0x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</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](#).

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
</tr>
</thead>
</table>
| 0   | 00  | 00  | 00  | 0000| SVE FFR write from predicate
| 1   | 00  | 00  | 1xxx| 0000| SVE FFR initialise
| 1   | 00  | 00  | x1xx| 0000| UNALLOCATED
| 1   | 00  | 00  | xx1x| 0000| UNALLOCATED
| 1   | 00  | 00  | xxx1| 0000| UNALLOCATED
| 0   | 00  | != 000| ! = 0000| UNALLOCATED
| != 00|     |      |      | UNALLOCATED
```

#### SVE FFR initialise

These instructions are under [SVE Write FFR](#).

```
<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>WRFFR</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
```
### SVE Integer Multiply-Add - Unpredicated

These instructions are under [SVE encodings](#).

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>SETFFR</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
```

### SVE integer dot product (unpredicated)

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td>SVE integer dot product (unpredicated)</td>
</tr>
<tr>
<td>!= 00000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
```

### SVE Multiply - Indexed

These instructions are under [SVE encodings](#).

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SDOT (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>UDOT (vectors)</td>
</tr>
</tbody>
</table>
```

### SVE integer dot product (indexed)

These instructions are under [SVE Multiply - Indexed](#).

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td>SVE integer dot product (indexed)</td>
</tr>
<tr>
<td>!= 00000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
```

### SVE Multiply - Indexed

These instructions are under [SVEMultiply - Indexed](#).

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SDOT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>UDOT (indexed) — 32-bit</td>
</tr>
</tbody>
</table>
```

---

*Page 2342*
SVE floating-point 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>size</td>
<td>U</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>01100000</td>
<td>00000000000ZnZd</td>
</tr>
</tbody>
</table>

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>size</td>
<td>op</td>
</tr>
<tr>
<td>0x</td>
<td>0</td>
</tr>
<tr>
<td>0x</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>rot</td>
</tr>
<tr>
<td>01100010</td>
<td>0001ZnZd</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>size</td>
<td>op</td>
</tr>
<tr>
<td>0x</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Zm</td>
</tr>
<tr>
<td>01100000</td>
<td>0100ZnZd</td>
</tr>
</tbody>
</table>

SVE floating-point compare vectors

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>op</td>
</tr>
<tr>
<td>0x</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Zm</td>
</tr>
<tr>
<td>01100100</td>
<td>0011ZnZd</td>
</tr>
</tbody>
</table>

Page 2343
### SVE floating-point arithmetic (unpredicated)

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADD (vectors, unpredicated)</td>
</tr>
<tr>
<td>001</td>
<td>FSUB (vectors, unpredicated)</td>
</tr>
<tr>
<td>010</td>
<td>FMUL (vectors, unpredicated)</td>
</tr>
<tr>
<td>011</td>
<td>FTSMUL</td>
</tr>
<tr>
<td>10X</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>FRECPS</td>
</tr>
<tr>
<td>111</td>
<td>FRSQRTS</td>
</tr>
</tbody>
</table>

### SVE Floating Point Arithmetic - Predicated

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>SVE floating-point arithmetic (predicated)</td>
</tr>
<tr>
<td>10 000</td>
<td>FTMAD</td>
</tr>
<tr>
<td>10 != 000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 0000</td>
<td>SVE floating-point arithmetic with immediate (predicated)</td>
</tr>
<tr>
<td>11 != 0000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE floating-point arithmetic (predicated)

These instructions are under **SVE Floating Point Arithmetic - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</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>
</tbody>
</table>
SVE floating-point arithmetic with immediate (predicated)

These instructions are under SVE Floating Point Arithmetic - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FADD (immediate)</td>
</tr>
<tr>
<td>0001</td>
<td>FSUB (immediate)</td>
</tr>
<tr>
<td>0100</td>
<td>FMUL (immediate)</td>
</tr>
<tr>
<td>0101</td>
<td>FSUBR (immediate)</td>
</tr>
<tr>
<td>0110</td>
<td>FMAXNM (immediate)</td>
</tr>
<tr>
<td>0111</td>
<td>FMINNM (immediate)</td>
</tr>
<tr>
<td>1000</td>
<td>FABD</td>
</tr>
<tr>
<td>1001</td>
<td>FSCALE</td>
</tr>
<tr>
<td>1010</td>
<td>FMULX</td>
</tr>
<tr>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1100</td>
<td>FDIVR</td>
</tr>
<tr>
<td>1101</td>
<td>FDIV</td>
</tr>
<tr>
<td>111x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Floating Point Unary Operations - Predicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>SVE floating-point round to integral value</td>
</tr>
<tr>
<td>010</td>
<td>SVE floating-point convert precision</td>
</tr>
<tr>
<td>011</td>
<td>SVE floating-point unary operations</td>
</tr>
<tr>
<td>10x</td>
<td>SVE integer convert to floating-point</td>
</tr>
<tr>
<td>11x</td>
<td>SVE integer convert to integer</td>
</tr>
</tbody>
</table>

SVE floating-point round to integral value

These instructions are under SVE Floating Point Unary Operations - Predicated.
### SVE floating-point convert precision

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

#### Instruction Details

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>FCVT — single-precision to half-precision</td>
</tr>
<tr>
<td>10</td>
<td>FCVT — half-precision to single-precision</td>
</tr>
<tr>
<td>11</td>
<td>FCVT — double-precision to half-precision</td>
</tr>
<tr>
<td>11</td>
<td>FCVT — half-precision to double-precision</td>
</tr>
<tr>
<td>11</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](#).

#### Instruction Details

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FRECPX</td>
</tr>
<tr>
<td>01</td>
<td>FSQRT</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE integer convert to floating-point

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

#### Instruction Details

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>01 00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 01 0</td>
<td><strong>SCVT</strong> — 16-bit to half-precision</td>
</tr>
<tr>
<td>01 01 1</td>
<td><strong>UCVT</strong> — 16-bit to half-precision</td>
</tr>
<tr>
<td>01 10 0</td>
<td><strong>SCVT</strong> — 32-bit to half-precision</td>
</tr>
<tr>
<td>01 10 1</td>
<td><strong>UCVT</strong> — 32-bit to half-precision</td>
</tr>
<tr>
<td>01 11 0</td>
<td><strong>SCVT</strong> — 64-bit to half-precision</td>
</tr>
<tr>
<td>01 11 1</td>
<td><strong>UCVT</strong> — 64-bit to half-precision</td>
</tr>
<tr>
<td>10 0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 10 0</td>
<td><strong>SCVT</strong> — 32-bit to single-precision</td>
</tr>
<tr>
<td>10 10 1</td>
<td><strong>UCVT</strong> — 32-bit to single-precision</td>
</tr>
<tr>
<td>10 11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 00 0</td>
<td><strong>SCVT</strong> — 32-bit to double-precision</td>
</tr>
<tr>
<td>11 00 1</td>
<td><strong>UCVT</strong> — 32-bit to double-precision</td>
</tr>
<tr>
<td>11 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 10 0</td>
<td><strong>SCVT</strong> — 64-bit to single-precision</td>
</tr>
<tr>
<td>11 10 1</td>
<td><strong>UCVT</strong> — 64-bit to single-precision</td>
</tr>
<tr>
<td>11 11 0</td>
<td><strong>SCVT</strong> — 64-bit to double-precision</td>
</tr>
<tr>
<td>11 11 1</td>
<td><strong>UCVT</strong> — 64-bit to double-precision</td>
</tr>
</tbody>
</table>

**SVE floating-point convert to integer**

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

---

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 01 0</td>
<td><strong>FCVTZS</strong> — half-precision to 16-bit</td>
</tr>
<tr>
<td>01 01 1</td>
<td><strong>FCVTZU</strong> — half-precision to 16-bit</td>
</tr>
<tr>
<td>01 10 0</td>
<td><strong>FCVTZS</strong> — half-precision to 32-bit</td>
</tr>
<tr>
<td>01 10 1</td>
<td><strong>FCVTZU</strong> — half-precision to 32-bit</td>
</tr>
<tr>
<td>01 11 0</td>
<td><strong>FCVTZS</strong> — half-precision to 64-bit</td>
</tr>
<tr>
<td>01 11 1</td>
<td><strong>FCVTZU</strong> — half-precision to 64-bit</td>
</tr>
<tr>
<td>10 0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 10 0</td>
<td><strong>FCVTZS</strong> — single-precision to 32-bit</td>
</tr>
<tr>
<td>10 10 1</td>
<td><strong>FCVTZU</strong> — single-precision to 32-bit</td>
</tr>
<tr>
<td>10 11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 00 0</td>
<td><strong>FCVTZS</strong> — double-precision to 32-bit</td>
</tr>
<tr>
<td>11 00 1</td>
<td><strong>FCVTZU</strong> — double-precision to 32-bit</td>
</tr>
<tr>
<td>11 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 10 0</td>
<td><strong>FCVTZS</strong> — single-precision to 64-bit</td>
</tr>
<tr>
<td>11 10 1</td>
<td><strong>FCVTZU</strong> — single-precision to 64-bit</td>
</tr>
<tr>
<td>11 11 0</td>
<td><strong>FCVTZS</strong> — double-precision to 64-bit</td>
</tr>
<tr>
<td>11 11 1</td>
<td><strong>FCVTZU</strong> — double-precision to 64-bit</td>
</tr>
</tbody>
</table>
SVE floating-point recursive reduction

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>FADDV</td>
</tr>
<tr>
<td>0 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1 x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0 0</td>
<td>FMAXNMV</td>
</tr>
<tr>
<td>1 0 1</td>
<td>FMINNMV</td>
</tr>
<tr>
<td>1 1 0</td>
<td>FMAXV</td>
</tr>
<tr>
<td>1 1 1</td>
<td>FMINV</td>
</tr>
</tbody>
</table>

SVE Floating Point Unary Operations - 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></td>
</tr>
<tr>
<td>0 0</td>
<td>SVE floating-point reciprocal estimate (unpredicated)</td>
</tr>
<tr>
<td>!= 0 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE floating-point reciprocal estimate (unpredicated)

These instructions are under SVE Floating Point Unary Operations - Unpredicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>0 xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0 x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1 0</td>
<td>FRECPE</td>
</tr>
<tr>
<td>1 1 1</td>
<td>FRSORTE</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>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SVE floating-point compare with zero</td>
</tr>
<tr>
<td>1</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>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>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</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>0</td>
</tr>
<tr>
<td>000</td>
<td>FADDA</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 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>op0</td>
<td>0</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**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>FMLA (vectors)</td>
</tr>
<tr>
<td>01</td>
<td>FMLLS (vectors)</td>
</tr>
<tr>
<td>10</td>
<td>FNMLA</td>
</tr>
<tr>
<td>11</td>
<td>FNMLLS</td>
</tr>
</tbody>
</table>
SVE floating-point multiply-accumulate writing multiplicand

These instructions are under SVE Floating Point Multiply-Add.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FMAD</td>
</tr>
<tr>
<td>01</td>
<td>FMSB</td>
</tr>
<tr>
<td>10</td>
<td>FNMA</td>
</tr>
<tr>
<td>11</td>
<td>FNMSB</td>
</tr>
</tbody>
</table>

SVE Memory - 32-bit Gather and Unsize Contiguous

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>x1 0xx 0</td>
</tr>
<tr>
<td>00</td>
<td>x1 0xx 1</td>
</tr>
<tr>
<td>01</td>
<td>x1 0xx 0</td>
</tr>
<tr>
<td>10</td>
<td>x1 0xx 0</td>
</tr>
<tr>
<td>11</td>
<td>0x 000 0</td>
</tr>
<tr>
<td>11</td>
<td>0x 000 1</td>
</tr>
<tr>
<td>11</td>
<td>0x 010</td>
</tr>
<tr>
<td>11</td>
<td>0x 0x1</td>
</tr>
<tr>
<td>11</td>
<td>1x 0xx 0</td>
</tr>
<tr>
<td>11</td>
<td>1x 0xx 1</td>
</tr>
<tr>
<td>! = 11</td>
<td>x0 0xx 0</td>
</tr>
<tr>
<td>00</td>
<td>10x</td>
</tr>
<tr>
<td>00</td>
<td>110 0</td>
</tr>
<tr>
<td>00</td>
<td>111 0</td>
</tr>
<tr>
<td>00</td>
<td>11x 1</td>
</tr>
<tr>
<td>01</td>
<td>1xx</td>
</tr>
<tr>
<td>1x</td>
<td>1xx</td>
</tr>
</tbody>
</table>

SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

These instructions are under SVE Memory - 32-bit Gather and Unsize Contiguous.

<table>
<thead>
<tr>
<th>Decode fields</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>Instruction Details</th>
<th>Decode fields</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U</td>
</tr>
<tr>
<td>LD1SH (scalar plus vector)</td>
<td>0</td>
</tr>
<tr>
<td>LDFF1SH (scalar plus vector)</td>
<td>0</td>
</tr>
<tr>
<td>LD1H (scalar plus vector)</td>
<td>1</td>
</tr>
<tr>
<td>LDFF1H (scalar plus vector)</td>
<td>1</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>Instruction Details</th>
<th>Decode fields</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U</td>
</tr>
<tr>
<td>UNALLOCATED</td>
<td>0</td>
</tr>
<tr>
<td>LD1W (scalar plus vector)</td>
<td>1</td>
</tr>
<tr>
<td>LDFF1W (scalar plus vector)</td>
<td>1</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>Instruction Details</th>
<th>Decode fields</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>msz</td>
</tr>
<tr>
<td>PRFB (scalar plus immediate)</td>
<td>00</td>
</tr>
<tr>
<td>PRFH (scalar plus immediate)</td>
<td>01</td>
</tr>
<tr>
<td>PRFW (scalar plus immediate)</td>
<td>10</td>
</tr>
<tr>
<td>PRFD (scalar plus immediate)</td>
<td>11</td>
</tr>
</tbody>
</table>

### SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

The following constraints also apply to this encoding: opc != 11 && opc != 11

<table>
<thead>
<tr>
<th>Instruction Details</th>
<th>Decode fields</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>opc</td>
</tr>
<tr>
<td>LD1SB (scalar plus vector)</td>
<td>00</td>
</tr>
<tr>
<td>LDFF1SB (scalar plus vector)</td>
<td>00</td>
</tr>
<tr>
<td>LD1B (scalar plus vector)</td>
<td>00</td>
</tr>
<tr>
<td>LDFF1B (scalar plus vector)</td>
<td>00</td>
</tr>
</tbody>
</table>
### SVE contiguous prefetch (scalar plus scalar)

These instructions are under **SVE Memory - 32-bit Gather and Unsized Contiguous**.

![Instruction Details Table]

### SVE 32-bit gather prefetch (vector plus immediate)

These instructions are under **SVE Memory - 32-bit Gather and Unsized Contiguous**.

![Instruction Details Table]

### SVE 32-bit gather load (vector plus immediate)

These instructions are under **SVE Memory - 32-bit Gather and Unsized Contiguous**.

![Instruction Details Table]
### SVE load and broadcast element

These instructions are under **SVE Memory - 32-bit Gather and Unsized Contiguous**.

<table>
<thead>
<tr>
<th>dtypeh</th>
<th>dtypel</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>LD1RB — 8-bit element</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>LD1RB — 16-bit element</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>LD1RB — 32-bit element</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>LD1RB — 64-bit element</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1RSW</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD1RH — 16-bit element</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LD1RH — 32-bit element</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>LD1RH — 64-bit element</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>LD1RSH — 64-bit element</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>LD1RSH — 32-bit element</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>LD1RW — 32-bit element</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>LD1RW — 64-bit element</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>LD1RSB — 64-bit element</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>LD1RSB — 32-bit element</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>LD1RSB — 16-bit element</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>LD1RD</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>op0 op1 op2</td>
<td>SVE contiguous non-temporal load (scalar plus immediate)</td>
</tr>
<tr>
<td>op0 op1 op2</td>
<td>SVE contiguous non-temporal load (scalar plus scalar)</td>
</tr>
<tr>
<td>op0 op1 op2</td>
<td>SVE load multiple structures (scalar plus immediate)</td>
</tr>
<tr>
<td>op0 op1 op2</td>
<td>SVE load multiple structures (scalar plus scalar)</td>
</tr>
<tr>
<td>op0 op1 op2</td>
<td>SVE load and broadcast quadword (scalar plus immediate)</td>
</tr>
<tr>
<td>op0 op1 op2</td>
<td>SVE contiguous load (scalar plus immediate)</td>
</tr>
<tr>
<td>op0 op1 op2</td>
<td>SVE contiguous load (scalar plus scalar)</td>
</tr>
<tr>
<td>op0 op1 op2</td>
<td>SVE contiguous non-fault load (scalar plus immediate)</td>
</tr>
</tbody>
</table>
### SVE load and broadcast quadword (scalar plus scalar)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>SVE load and broadcast quadword (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE contiguous load (scalar plus scalar)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>SVE contiguous load (scalar plus scalar)</td>
</tr>
<tr>
<td>011</td>
<td>SVE contiguous first-fault load (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE contiguous 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>
<tr>
<td>11</td>
<td>LDNT1D (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></td>
</tr>
<tr>
<td>00</td>
<td>LDNT1B (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>LDNT1H (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>LDNT1W (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>LDNT1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### 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 opc</td>
<td></td>
</tr>
<tr>
<td>00 01</td>
<td>LD2B (scalar plus immediate)</td>
</tr>
<tr>
<td>00 10</td>
<td>LD3B (scalar plus immediate)</td>
</tr>
<tr>
<td>00 11</td>
<td>LD4B (scalar plus immediate)</td>
</tr>
<tr>
<td>01 01</td>
<td>LD2H (scalar plus immediate)</td>
</tr>
<tr>
<td>01 10</td>
<td>LD3H (scalar plus immediate)</td>
</tr>
<tr>
<td>01 11</td>
<td>LD4H (scalar plus immediate)</td>
</tr>
<tr>
<td>10 01</td>
<td>LD2W (scalar plus immediate)</td>
</tr>
<tr>
<td>10 10</td>
<td>LD3W (scalar plus immediate)</td>
</tr>
<tr>
<td>10 11</td>
<td>LD4W (scalar plus immediate)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 & opc != 0
### SVE load multiple structures (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

- The following constraints also apply to this encoding: \( \text{opc} \neq 00 \land \text{opc} \neq 00 \)

### 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**.

<table>
<thead>
<tr>
<th>Decode fields dtype</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LD1B (scalar plus immediate) — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LD1B (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LD1B (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LD1B (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LD1SW (scalar plus immediate)</td>
</tr>
<tr>
<td>0101</td>
<td>LD1H (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LD1H (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LD1H (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LD1SH (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td>LD1SH (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td>LD1W (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LD1W (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td>LD1SB (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LD1SB (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LD1SB (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td>LD1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

### SVE load and broadcast quadword (scalar plus scalar)

These instructions are under **SVE Memory - Contiguous Load**.

<table>
<thead>
<tr>
<th>Decode fields 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>
### SVE contiguous load (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

<table>
<thead>
<tr>
<th>Msz</th>
<th>Sz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>LD1RQB (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1ROH (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>LD1ROW (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>LD1ROD (scalar plus scalar)</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>Msz</th>
<th>Sz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>LD1RQB (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1ROH (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>LD1ROW (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>LD1ROD (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE contiguous first-fault load (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

<table>
<thead>
<tr>
<th>Msz</th>
<th>Sz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>LD1F1B (scalar plus scalar) — 8-bit element</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1F1B (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>LD1F1B (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>LD1F1B (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1F1SW (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD1F1H (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LD1F1H (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>LD1F1H (scalar plus scalar) — 64-bit element</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>Msz</th>
<th>Sz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>LD1F1B (scalar plus scalar) — 8-bit element</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1F1B (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>LD1F1B (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>LD1F1B (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1F1SW (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD1F1H (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LD1F1H (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>LD1F1H (scalar plus scalar) — 64-bit element</td>
</tr>
</tbody>
</table>
### SVE Memory - 64-bit Gather

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td><code>LDFF1SH</code> (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td><code>LDFF1SH</code> (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td><code>LDFF1W</code> (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td><code>LDFF1W</code> (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td><code>LDFF1SB</code> (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td><code>LDFF1SB</code> (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td><code>LDFF1SB</code> (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td><code>LDFF1D</code> (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 01 0xx 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 11 1xx 0</td>
<td>SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)</td>
</tr>
<tr>
<td>00 11 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 x1 0xx 0</td>
<td>SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)</td>
</tr>
<tr>
<td>!= 00 11 1xx</td>
<td>SVE 64-bit gather load (scalar plus 64-bit scaled offsets)</td>
</tr>
<tr>
<td>!= 00 x1 0xx</td>
<td>SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)</td>
</tr>
<tr>
<td>00 10x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 111 0</td>
<td>SVE 64-bit gather prefetch (vector plus immediate)</td>
</tr>
<tr>
<td>00 111 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 1xx</td>
<td>SVE 64-bit gather load (vector plus immediate)</td>
</tr>
<tr>
<td>10 1xx</td>
<td>SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)</td>
</tr>
<tr>
<td>x0 0xx</td>
<td>SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)</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>00</td>
<td><code>PRFB</code> (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td><code>PRFH</code> (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td><code>PRFW</code> (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td><code>PRFD</code> (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).
Top-level encodings for A64

<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>PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

**SVE 64-bit gather load (scalar plus 64-bit scaled offsets)**

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

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](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 & opc != 00
### 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 11 0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11 1 1</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather load (vector plus 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></td>
</tr>
<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 64-bit gather load (scalar plus 64-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></td>
</tr>
<tr>
<td>00 0 0 0</td>
<td>LD1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td>LDFF1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00 1 0 0</td>
<td>LD1B (vector plus immediate)</td>
</tr>
<tr>
<td>00 1 1 0</td>
<td>LDFF1B (vector plus immediate)</td>
</tr>
<tr>
<td>01 0 0 0</td>
<td>LD1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01 0 1 0</td>
<td>LDFF1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01 1 0 0</td>
<td>LD1H (vector plus immediate)</td>
</tr>
<tr>
<td>01 1 1 0</td>
<td>LDFF1H (vector plus immediate)</td>
</tr>
<tr>
<td>10 0 0 0</td>
<td>LD1SW (vector plus immediate)</td>
</tr>
<tr>
<td>10 0 1 0</td>
<td>LDFF1SW (vector plus immediate)</td>
</tr>
<tr>
<td>10 1 0 0</td>
<td>LD1W (vector plus immediate)</td>
</tr>
<tr>
<td>10 1 1 0</td>
<td>LDFF1W (vector plus immediate)</td>
</tr>
<tr>
<td>11 0 0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 0 0</td>
<td>LD1D (vector plus immediate)</td>
</tr>
<tr>
<td>11 1 1 0</td>
<td>LDFF1D (vector plus immediate)</td>
</tr>
</tbody>
</table>
### SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 1 0 0 0 1 0 | msz | xs | 0 | Zm | 0 | U | ff | Pg | Rn | Zt |
```

### SVE Memory - Contiguous Store and Unsized Contiguous

These instructions are under [SVE encodings](#).

```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
| 11 10 01 0 | op0 | 0 | op1 | 0 | op2 |
```

---

### Instructions Details

- **LD1SB** (scalar plus vector)
- **LDFF1SB** (scalar plus vector)
- **LD1B** (scalar plus vector)
- **LDFF1B** (scalar plus vector)
- **LD1SH** (scalar plus vector)
- **LDFF1SH** (scalar plus vector)
- **LD1H** (scalar plus vector)
- **LDFF1H** (scalar plus vector)
- **LD1SW** (scalar plus vector)
- **LDFF1SW** (scalar plus vector)
- **LD1W** (scalar plus vector)
- **LDFF1W** (scalar plus vector)
- **LD1D** (scalar plus vector)
- **LDFF1D** (scalar plus vector)
- **UNALLOCATED**

---

[1]: #/SVE Memory - 64-bit Gather
[2]: #/SVE encodings

---

Page 2361
SVE contiguous store (scalar plus scalar)

These instructions are under SVE Memory - Contiguous Store and Unsized Contiguous.

The following constraints also apply to this encoding: opc != 110 && opc != 110

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>ST1B (scalar plus scalar)</td>
</tr>
<tr>
<td></td>
<td>ST1H (scalar plus scalar)</td>
</tr>
<tr>
<td>10x</td>
<td>ST1W (scalar plus scalar)</td>
</tr>
<tr>
<td>111 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111 1</td>
<td>ST1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE Memory - Non-temporal and Multi-register Store

These instructions are under SVE encodings.

The following constraints also apply to this encoding: opc != 110 & op1 != 0

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td>SVE contiguous non-temporal store (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>SVE store multiple structures (scalar plus scalar)</td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE contiguous non-temporal store (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</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>
SVE store multiple structures (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

The following constraints also apply to this encoding: opc != 00 && opc != 00

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 01</td>
<td>ST2B (scalar plus scalar)</td>
</tr>
<tr>
<td>00 10</td>
<td>ST3B (scalar plus scalar)</td>
</tr>
<tr>
<td>00 11</td>
<td>ST4B (scalar plus scalar)</td>
</tr>
<tr>
<td>01 01</td>
<td>ST2H (scalar plus scalar)</td>
</tr>
<tr>
<td>01 10</td>
<td>ST3H (scalar plus scalar)</td>
</tr>
<tr>
<td>01 11</td>
<td>ST4H (scalar plus scalar)</td>
</tr>
<tr>
<td>10 01</td>
<td>ST2W (scalar plus scalar)</td>
</tr>
<tr>
<td>10 10</td>
<td>ST3W (scalar plus scalar)</td>
</tr>
<tr>
<td>10 11</td>
<td>ST4W (scalar plus scalar)</td>
</tr>
<tr>
<td>11 01</td>
<td>ST2D (scalar plus scalar)</td>
</tr>
<tr>
<td>11 10</td>
<td>ST3D (scalar plus scalar)</td>
</tr>
<tr>
<td>11 11</td>
<td>ST4D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE Memory - Scatter with Optional Sign Extend

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)</td>
</tr>
<tr>
<td>10</td>
<td>SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)</td>
</tr>
<tr>
<td>11</td>
<td>SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)</td>
</tr>
</tbody>
</table>

SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

<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>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

Page 2363
**SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled 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>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

**SVE 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 32-bit scatter store (scalar plus 32-bit scaled offsets)**

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

<table>
<thead>
<tr>
<th>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 Memory - Scatter**

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>
### SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)

These instructions are under [SVE Memory - Scatter](#).

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)

These instructions are under [SVE Memory - Scatter](#).

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit scatter store (vector plus immediate)

These instructions are under [SVE Memory - Scatter](#).

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (vector plus immediate)</td>
</tr>
</tbody>
</table>

### SVE 32-bit scatter store (vector plus immediate)

These instructions are under [SVE Memory - Scatter](#).

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
**SVE Memory - Contiguous Store with Immediate Offset**

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>SVE contiguous non-temporal store (scalar plus immediate)</td>
</tr>
<tr>
<td>!= 00</td>
<td>1</td>
<td>SVE store multiple structures (scalar plus immediate)</td>
</tr>
<tr>
<td>0</td>
<td>SVE contiguous store (scalar plus immediate)</td>
<td></td>
</tr>
</tbody>
</table>

### SVE contiguous non-temporal store (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

<table>
<thead>
<tr>
<th>msz</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>

### SVE store multiple structures (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST2B (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>ST3B (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>ST4B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST2H (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST3H (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST4H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST2W (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST3W (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST4W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST2D (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST3D (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST4D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST2B (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>ST3B (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>ST4B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST2H (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST3H (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST4H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST2W (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST3W (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST4W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST2D (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST3D (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST4D (scalar plus immediate)</td>
</tr>
</tbody>
</table>
### Top-level encodings for A64

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

### Data Processing -- Immediate

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>PC-rel. addressing</td>
</tr>
<tr>
<td>010</td>
<td>Add/subtract (immediate)</td>
</tr>
<tr>
<td>011</td>
<td>Add/subtract (immediate, with tags)</td>
</tr>
<tr>
<td>100</td>
<td>Logical (immediate)</td>
</tr>
<tr>
<td>101</td>
<td>Move wide (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>Bitfield</td>
</tr>
<tr>
<td>111</td>
<td>Extract</td>
</tr>
</tbody>
</table>

### PC-rel. addressing

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADR</td>
</tr>
<tr>
<td>1</td>
<td>ADRP</td>
</tr>
</tbody>
</table>

### Add/subtract (immediate)

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>ADD (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>ADDS (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>SUB (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>SUBS (immediate) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ADD (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ADDS (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUB (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUBS (immediate) — 64-bit</td>
</tr>
</tbody>
</table>
Add/subtract (immediate, with tags)

These instructions are under Data Processing -- Immediate.

Logical (immediate)

These instructions are under Data Processing -- Immediate.

Move wide (immediate)

These instructions are under Data Processing -- Immediate.
### Bitfield

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>hw</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td></td>
<td>MOVK — 64-bit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>N</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</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**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op2</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>N</th>
<th>o0</th>
<th>Rm</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>op1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>imms</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>EXTR — 32-bit</td>
<td></td>
</tr>
</tbody>
</table>

### 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>00xxx</td>
<td></td>
<td>Exception generation</td>
</tr>
<tr>
<td>110</td>
<td>010000110001</td>
<td></td>
<td>System instructions with register argument</td>
</tr>
<tr>
<td>110</td>
<td>0100001100010</td>
<td>1111</td>
<td>Hints</td>
</tr>
</tbody>
</table>

---

**Top-level encodings for A64**
Conditional branch (immediate)

These instructions are under Branches, Exception Generating and System 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
0 1 0 1 0 1 0 0 | o1 imm19 | o0 | cond
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1 0 0 0</td>
<td>B.cond</td>
</tr>
<tr>
<td>o1 0 1 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>o1 1 0 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Exception generation

These instructions are under Branches, Exception Generating and System 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 | opc imm16 | op2 | LL
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc 001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 1xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 000 000 00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 000 000 01</td>
<td>SVC</td>
</tr>
<tr>
<td>opc 000 000 10</td>
<td>HVC</td>
</tr>
<tr>
<td>opc 000 000 11</td>
<td>SMC</td>
</tr>
<tr>
<td>opc 001 000 x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 001 000 00</td>
<td>BRK</td>
</tr>
<tr>
<td>opc 001 000 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 010 000 x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 010 000 00</td>
<td>HLT</td>
</tr>
<tr>
<td>opc 010 000 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 011 000 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 011 000 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 100 000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 101 000 00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 101 000 01</td>
<td>DCPS1</td>
</tr>
<tr>
<td>opc 101 000 10</td>
<td>DCPS2</td>
</tr>
<tr>
<td>opc 101 000 11</td>
<td>DCPS3</td>
</tr>
<tr>
<td>opc 110 000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### System instructions with register argument

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc op2 LL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### Hints

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>Decode fields op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!= 0000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0000 000</td>
<td>WFE T</td>
<td>FEAT WFxT</td>
</tr>
<tr>
<td>0000 001</td>
<td>WFI T</td>
<td>FEAT WFxT</td>
</tr>
<tr>
<td>0000 01x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0000 1xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 000</td>
<td>NOP</td>
<td>-</td>
</tr>
<tr>
<td>0000 001</td>
<td>YIELD</td>
<td>-</td>
</tr>
<tr>
<td>0000 010</td>
<td>WFE</td>
<td>-</td>
</tr>
<tr>
<td>0000 011</td>
<td>WFI</td>
<td>-</td>
</tr>
<tr>
<td>0000 100</td>
<td>SEV</td>
<td>-</td>
</tr>
<tr>
<td>0000 101</td>
<td>SEVL</td>
<td>-</td>
</tr>
<tr>
<td>0000 110</td>
<td>DGH</td>
<td>FEAT DGH</td>
</tr>
<tr>
<td>0000 111</td>
<td>XPACD,XPACI,XPACLRI</td>
<td></td>
</tr>
<tr>
<td>0001 000</td>
<td>PACIA,PACIA1716,PACIASP,PACIAZ,PACIAZ→PACIA1716</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0001 010</td>
<td>PACIB,PACIB1716,PACIBSP,PACIBZ,PACIZB→PACIB1716</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0001 100</td>
<td>AUTIA,AUTIA1716,AUTIASP,AUTIAZ,AUTIZA→AUTIA1716</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0001 110</td>
<td>AUTIB,AUTIB1716,AUTIBSP,AUTIBZ,AUTIZB→AUTIB1716</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0010 000</td>
<td>ESB</td>
<td>FEAT RAS</td>
</tr>
<tr>
<td>0010 001</td>
<td>PSB,CSYNC</td>
<td>FEAT SPE</td>
</tr>
<tr>
<td>0010 010</td>
<td>TSB,CSYNC</td>
<td>FEAT TRF</td>
</tr>
<tr>
<td>0010 100</td>
<td>CSDB</td>
<td>-</td>
</tr>
<tr>
<td>0011 000</td>
<td>PACIA,PACIA1716,PACIASP,PACIAZ,PACIAZ→PACIAZ</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0011 001</td>
<td>PACIA,PACIA1716,PACIASP,PACIAZ,PACIAZ→PACIASP</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0011 010</td>
<td>PACIB,PACIB1716,PACIBSP,PACIBZ,PACIZB→PACIBZ</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0011 011</td>
<td>PACIB,PACIB1716,PACIBSP,PACIBZ,PACIZB→PACIBSP</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0011 100</td>
<td>AUTIA,AUTIA1716,AUTIASP,AUTIAZ,AUTIZA→AUTIAZ</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0011 101</td>
<td>AUTIA,AUTIA1716,AUTIASP,AUTIAZ,AUTIZA→AUTIASP</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0011 110</td>
<td>AUTIB,AUTIB1716,AUTIBSP,AUTIBZ,AUTIZB→AUTIBZ</td>
<td>FEAT PAuth</td>
</tr>
<tr>
<td>0011 111</td>
<td>AUTIB,AUTIB1716,AUTIBSP,AUTIBZ,AUTIZB→AUTIBSP</td>
<td>FEAT PAuth</td>
</tr>
</tbody>
</table>
### Barriers

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>xx0</td>
<td>BTI</td>
<td>FEAT_BTI</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>!= 11111</td>
<td>11111</td>
<td>CLREX</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>11111</td>
<td></td>
<td>DMB</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>11111</td>
<td></td>
<td>ISB</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>!= 11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SB</td>
<td></td>
</tr>
<tr>
<td>xx0x</td>
<td>001</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>xx10</td>
<td>001</td>
<td>11111</td>
<td>DSB — Memory nXS barrier</td>
<td>FEAT XS</td>
</tr>
<tr>
<td>xx11</td>
<td>001</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>!= 0x00</td>
<td>100</td>
<td>11111</td>
<td>DSB — memory barrier</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>100</td>
<td>11111</td>
<td>SSBB</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>100</td>
<td>11111</td>
<td>PSSBB</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### System instructions

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>op1</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>!= 11111</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11111</td>
<td>MSR (immediate)</td>
</tr>
<tr>
<td>0000</td>
<td>000</td>
<td>11111</td>
<td>CFINV</td>
</tr>
<tr>
<td>0000</td>
<td>001</td>
<td>11111</td>
<td>XAFLAG</td>
</tr>
<tr>
<td>0000</td>
<td>010</td>
<td>11111</td>
<td>AXFLAG</td>
</tr>
</tbody>
</table>
System register move

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SYS</td>
</tr>
<tr>
<td>1</td>
<td>SYSL</td>
</tr>
</tbody>
</table>

Unconditional branch (register)

These instructions are under Branches, Exception Generating and System instructions.

<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>

```
0000 11111 000000 000000 UNALLOCATED -
0000 11111 000000 000000 BR -
0000 11111 000001 000000 UNALLOCATED -
0000 11111 000010 11111 BRAA, BRAAZ, BRAB, BRABZ — key_A, zero modifier FEAT_PAuth
0000 11111 000011 11111 BRAA, BRAAZ, BRAB, BRABZ — key_B, zero modifier FEAT_PAuth
0000 11111 0001xx UNALLOCATED -
0000 11111 001xxx UNALLOCATED -
0000 11111 01xxxx UNALLOCATED -
0000 11111 1xxxxx UNALLOCATED -
0001 11111 000000 000000 BLR -
0001 11111 000001 000000 UNALLOCATED -
0001 11111 000010 11111 BLRAA, BLRAAZ, BLRAB, BLRABZ — key_A, zero modifier FEAT_PAuth
0001 11111 000011 11111 UNALLOCATED -
```
<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011 1111 000011 000001 11111</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 1111 0001xx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 1111 001xxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 1111 01xxxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 1111 1xxxxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 000000 != 00000 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 000000 00000 RET</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 000001 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 000010 != 11111 != 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 000010 11111 11111 RETAA, RETAB — RETAA FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 000011 != 11111 != 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 000011 11111 11111 RETAA, RETAB — RETAB FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 001xx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 001xxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 01xxxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 11111 1xxxxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000000 != 11111 != 00000 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000000 != 11111 00000 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000000 11111 != 00000 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000001 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000010 != 11111 != 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000010 != 11111 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000010 11111 != 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000010 11111 11111 ERETA, ERETAB — ERETA FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000011 != 11111 != 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000011 != 11111 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000011 11111 != 11111 UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 000011 11111 11111 ERETA, ERETAB — ERETA FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 001xx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 001xxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 01xxxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 11111 1xxxxx UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Unconditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th>op</th>
<th>0 0 1 0 1</th>
</tr>
</thead>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>BL</td>
</tr>
</tbody>
</table>

### Compare and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th>sf</th>
<th>0 1 1 0 1 0</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm19</td>
<td></td>
<td>Rt</td>
</tr>
</tbody>
</table>
### Test and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>CBZ — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>CBNZ — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CBZ — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CBNZ — 64-bit</td>
</tr>
</tbody>
</table>

### Loads and Stores

These instructions are under the [top-level](#).

<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>0x00</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>000000</td>
<td>Advanced SIMD load/store multiple structures</td>
</tr>
<tr>
<td>0x00</td>
<td>0</td>
<td>01</td>
<td>XXXX</td>
<td></td>
<td>Advanced SIMD load/store multiple structures (post-indexed)</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>0X</td>
<td>1XXX</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>10</td>
<td>X0000</td>
<td></td>
<td>Advanced SIMD load/store single structure</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
<td>Advanced SIMD load/store single structure (post-indexed)</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>X0</td>
<td>X1XXX</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>X0</td>
<td>XX1XX</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>X0</td>
<td>XXX1XX</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>X0</td>
<td>XXXX1X</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>X0</td>
<td>XXXX1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>1X</td>
<td>1XXX</td>
<td></td>
<td>Load/store memory tags</td>
</tr>
<tr>
<td>1x00</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>xx00</td>
<td>0</td>
<td>0X</td>
<td></td>
<td></td>
<td>Load/store exclusive</td>
</tr>
<tr>
<td>xx01</td>
<td>0</td>
<td>1X</td>
<td>0XXX</td>
<td>00</td>
<td>LDAPR/STLR (unscaled immediate)</td>
</tr>
<tr>
<td>xx01</td>
<td>0</td>
<td>0X</td>
<td></td>
<td></td>
<td>Load register (literal)</td>
</tr>
<tr>
<td>xx10</td>
<td>0</td>
<td>00</td>
<td></td>
<td></td>
<td>Load/store no-allocate pair (offset)</td>
</tr>
<tr>
<td>xx10</td>
<td>0</td>
<td>01</td>
<td></td>
<td></td>
<td>Load/store register pair (post-indexed)</td>
</tr>
<tr>
<td>xx10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>Load/store register pair (offset)</td>
</tr>
<tr>
<td>xx10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>Load/store register pair (pre-indexed)</td>
</tr>
<tr>
<td>xx11</td>
<td>0</td>
<td>0X</td>
<td>0XXX</td>
<td>00</td>
<td>Load/store register (unscaled immediate)</td>
</tr>
<tr>
<td>xx11</td>
<td>0</td>
<td>0X</td>
<td>0XXX</td>
<td>01</td>
<td>Load/store register (immediate post-indexed)</td>
</tr>
<tr>
<td>xx11</td>
<td>0</td>
<td>0X</td>
<td>0XXX</td>
<td>10</td>
<td>Load/store register (unprivileged)</td>
</tr>
<tr>
<td>xx11</td>
<td>0</td>
<td>0X</td>
<td>0XXX</td>
<td>11</td>
<td>Load/store register (immediate pre-indexed)</td>
</tr>
<tr>
<td>xx11</td>
<td>0</td>
<td>0X</td>
<td>1XXX</td>
<td>00</td>
<td>Atomic memory operations</td>
</tr>
</tbody>
</table>
### Advanced SIMD load/store multiple structures

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000</td>
<td>ST4 (multiple structures)</td>
</tr>
<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>0 11xx</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>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>L Rm 0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>L Rm 0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>L Rm 0101</td>
<td>UNALLOCATED</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</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td>ST3 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x0</td>
<td></td>
<td></td>
<td>ST1 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x0</td>
<td></td>
<td></td>
<td>ST3 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>00</td>
<td></td>
<td></td>
<td>ST1 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>01</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>00</td>
<td></td>
<td></td>
<td>ST3 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>01</td>
<td></td>
<td></td>
<td>ST3 (single structure) — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td>ST2 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td>ST4 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x0</td>
<td></td>
<td></td>
<td>ST2 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x0</td>
<td></td>
<td></td>
<td>ST4 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>00</td>
<td></td>
<td></td>
<td>ST2 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>01</td>
<td></td>
<td></td>
<td>ST2 (single structure) — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>00</td>
<td></td>
<td></td>
<td>ST4 (single structure) — 32-bit</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>01</td>
<td></td>
<td></td>
<td>ST4 (single structure) — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td>LD1 (single structure) — 8-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td>LD3 (single structure) — 8-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x0</td>
<td></td>
<td></td>
<td>LD1 (single structure) — 16-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x0</td>
<td></td>
<td></td>
<td>LD3 (single structure) — 16-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>00</td>
<td></td>
<td></td>
<td>LD1 (single structure) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>01</td>
<td></td>
<td></td>
<td>LD1 (single structure) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>00</td>
<td></td>
<td></td>
<td>LD3 (single structure) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>01</td>
<td></td>
<td></td>
<td>LD3 (single structure) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</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>Decode fields</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>001</td>
<td></td>
<td></td>
<td>ST3 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST1 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST3 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST1 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td></td>
<td>ST1 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td></td>
<td>ST3 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td></td>
<td>ST3 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1 x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>000</td>
<td></td>
<td></td>
<td>ST2 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>001</td>
<td></td>
<td></td>
<td>ST4 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST2 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST4 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST2 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>01</td>
<td></td>
<td>ST2 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td></td>
<td>ST4 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>01</td>
<td></td>
<td>ST4 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>000</td>
<td></td>
<td></td>
<td>ST2 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>001</td>
<td></td>
<td></td>
<td>ST4 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST2 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST4 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST2 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td></td>
<td>ST2 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td></td>
<td>ST4 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td></td>
<td>ST4 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1 x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1 01</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>1 11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>1 x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>000</td>
<td></td>
<td></td>
<td>LD1 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>001</td>
<td></td>
<td></td>
<td>LD3 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>LD1 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>LD3 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>LD1 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>01</td>
<td></td>
<td>LD1 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td></td>
<td>LD3 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Rm</td>
<td>opcode</td>
<td>S</td>
<td>size</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>----</td>
<td>--------</td>
<td>---</td>
<td>------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>101</td>
<td>001</td>
<td>LD3 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>LD1R — register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD3R — register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>000</td>
<td></td>
<td>LD1 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>001</td>
<td></td>
<td>LD3 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>LD1 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>LD3 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>100</td>
<td>0</td>
<td>LD1 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>LD3 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>101</td>
<td>0</td>
<td>LD3 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>LD1 (single structure) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>LD1R — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD3R — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>010</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>011</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>100</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>100</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>100</td>
<td>1x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>101</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>101</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>101</td>
<td>1x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>110</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>111</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>000</td>
<td></td>
<td>LD2 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>001</td>
<td></td>
<td>LD4 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>LD2 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>LD4 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>LD2 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td>LD2 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>LD4 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>LD2R — register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD4R — register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>000</td>
<td></td>
<td>LD2 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>001</td>
<td></td>
<td>LD4 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>LD2 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>LD4 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>LD2 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td>LD2 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>LD4 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>LD2R — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD4R — immediate offset</td>
</tr>
</tbody>
</table>

Load/store memory tags

These instructions are under **Loads and Stores**.
### decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>STG — post-index</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>STG — signed offset</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>STG — pre-index</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>00</td>
<td>0000000000</td>
<td>STZGM</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LDG</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>STZG — post-index</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>STZG — signed offset</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>STZG — pre-index</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>ST2G — post-index</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>ST2G — signed offset</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>ST2G — pre-index</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>10 ! = 000000000</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0000000000</td>
<td>STGM</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>STZ2G — post-index</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>STZ2G — signed offset</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>STZ2G — pre-index</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>11 ! = 000000000</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0000000000</td>
<td>LDGM</td>
<td>FEAT_MTE</td>
</tr>
</tbody>
</table>

### load/store exclusive

These instructions are under [Loads and Stores](#).

### Instruction Details

<table>
<thead>
<tr>
<th>size</th>
<th>o2</th>
<th>Decode fields</th>
<th>Rt2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STXRB</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLXRB</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASP</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASPL</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>LDXRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>LDAXRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASP</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASPL</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>STLLRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>STLRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASLB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>LDLARB</td>
</tr>
<tr>
<td>size</td>
<td>o2</td>
<td>Decode fields</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>00</td>
<td>1</td>
<td>1 1 1 0</td>
<td>LDARB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1 1 1 1</td>
<td>CASB, CASAB, CASALB, CASLB — CASAB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1 1 1 1</td>
<td>CASB, CASAB, CASALB, CASLB — CASALB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0 0 0 0</td>
<td>STXRH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0 0 0 1</td>
<td>STLXRH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0 1 1 0</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASP</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0 1 1 1</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPL</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1 1 1 0</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPA</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1 1 1 1</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0 1 1 0</td>
<td>STLLRH</td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0 1 1 1</td>
<td>STLLRH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0 1 1 0</td>
<td>CASH, CASAH, CASALH, CASLH — CASH</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0 1 1 1</td>
<td>CASH, CASAH, CASALH, CASLH — CASALH</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1 1 1 0</td>
<td>LDLARH</td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1 1 1 1</td>
<td>LDLARH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0 1 1 0</td>
<td>LDAXR — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0 1 1 1</td>
<td>LDAXP — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1 1 1 0</td>
<td>LDAXR — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1 1 1 1</td>
<td>LDAXP — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0 0 0 0</td>
<td>STXR — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0 0 0 1</td>
<td>STLXR — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0 1 1 0</td>
<td>STXP — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1 1 1 0</td>
<td>LDAR — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1 1 1 1</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASA</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0 1 1 0</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASL</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1 0 1 0</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASL</td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1 1 1 0</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASA</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1 1 1 1</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASAL</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
</tbody>
</table>

Top-level encodings for A64
Top-level encodings for A64

### LDAPR/STLR (unscaled immediate)
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>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>LDAXP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLLR — 64-bit</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>STLR — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 64-bit CAS</td>
<td>FEAT_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>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDLAR — 64-bit</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></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>FEAT_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>FEAT_LSE</td>
</tr>
</tbody>
</table>

### Load register (literal)
These instructions are under Loads and Stores.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>STLURB</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>LDAPURB</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>LDAPURSB — 64-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>LDAPURSB — 32-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>STLURH</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LDAPURH</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LDAPURSH — 64-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>LDAPURSH — 32-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>STLUR — 32-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>LDAPUR — 32-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>LDAPURSW</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>STLUR — 64-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>LDAPUR — 64-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Decoders:
- **size**
- **opc**
- **Instruction Details**
- **Architecture Version**
## Load/store no-allocate pair (offset)

These instructions are under [Loads and Stores](#).

```
31 30 29 28 27 26 25 24 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 V 0 0 0 L imm7  Rt2  Rn  Rt
```

## Load/store register pair (post-indexed)

These instructions are under [Loads and Stores](#).

```
31 30 29 28 27 26 25 24 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 V 0 0 1 L imm7  Rt2  Rn  Rt
```
**Load/store register pair (offset)**

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Load/store register pair (pre-indexed)**

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Load/store register (unscaled immediate)**

These instructions are under [Loads and Stores](#).
## 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>x1 1 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 00</td>
<td>STURB</td>
</tr>
<tr>
<td>00 0 01</td>
<td>LDURB</td>
</tr>
<tr>
<td>00 0 10</td>
<td>LDURSB — 64-bit</td>
</tr>
<tr>
<td>00 0 11</td>
<td>LDURSB — 32-bit</td>
</tr>
<tr>
<td>00 1 00</td>
<td>STUR (SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00 1 01</td>
<td>LDUR (SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00 1 10</td>
<td>STUR (SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00 1 11</td>
<td>LDUR (SIMD&amp;FP) — 128-bit</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 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>PRFUM</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>
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>

#### Load/store register (unprivileged)

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>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>LDTRSW</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STTR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDTR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### Load/store register (immediate pre-indexed)

These instructions are under [Loads and Stores](#).
### Atomic memory operations

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>Decoded fields</td>
<td>o3</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>----------------</td>
<td>----</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td></td>
<td>1 011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td></td>
<td>1 100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td></td>
<td>1 101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>0 000</td>
<td>LDADDB, LDADDAB, LDADDLAB, LDADDLB — LDADDB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>0 001</td>
<td>LDCLRAB, LDCLRALB, LDCLRAB, LDCLRLB — LDCLRAB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>0 010</td>
<td>LDEORAB, LDEORALB, LDEORLB — LDEORAB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>0 011</td>
<td>LDSETAB, LDSETALB, LDSETLB — LDSETAB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>0 100</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>0 101</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>0 110</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>0 111</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>1 000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>1 001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>1 010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td></td>
<td>1 011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>0 000</td>
<td>LDADDB, LDADDAB, LDADDLAB, LDADDLB — LDADDB</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>0 001</td>
<td>LDCLRAB, LDCLRALB, LDCLRAB, LDCLRLB — LDCLRAB</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>0 010</td>
<td>LDEORAB, LDEORALB, LDEORLB — LDEORAB</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>0 011</td>
<td>LDSETAB, LDSETALB, LDSETLB — LDSETAB</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>0 100</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXB</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>0 101</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINB</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>0 110</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXB</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>0 111</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINB</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td></td>
<td>1 000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td></td>
<td>0 000</td>
<td>LDADDB, LDADDAB, LDADDLAB, LDADDLB — LDADDB</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td></td>
<td>0 001</td>
<td>LDCLRAB, LDCLRALB, LDCLRAB, LDCLRLB — LDCLRAB</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td></td>
<td>0 010</td>
<td>LDEORAB, LDEORALB, LDEORLB — LDEORAB</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td></td>
<td>0 011</td>
<td>LDSETAB, LDSETALB, LDSETLB — LDSETAB</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td></td>
<td>0 100</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXB</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td></td>
<td>0 101</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINB</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
## Instruction Details

### Architecture

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>V</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

### Load/store register (register offset)

These instructions are under Loads and Stores.

---

**Load/store register (register offset)**

These instructions are under **Loads and Stores**.

![Table](image)

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>V</td>
<td>0</td>
<td>opc</td>
<td>Rm</td>
</tr>
</tbody>
</table>

---

**Instruction Details**

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>! = 011</td>
<td>STRB (register) — extended register</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>! = 011</td>
<td>STRB (register) — shifted register</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>! = 011</td>
<td>LDRB (register) — extended register</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>! = 011</td>
<td>LDRB (register) — shifted register</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>! = 011</td>
<td>LDRSB (register) — 64-bit with extended register offset</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>! = 011</td>
<td>LDRSB (register) — 64-bit with shifted register offset</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>! = 011</td>
<td>LDRSB (register) — 32-bit with extended register offset</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>! = 011</td>
<td>LDRSB (register) — 32-bit with shifted register offset</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>! = 011</td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>! = 011</td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>! = 011</td>
<td>LDR (register, SIMD&amp;FP)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>! = 011</td>
<td>LDRH (register)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>STRH (register)</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>LDRH (register)</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>LDRSH (register)</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>LDRSH (register)</td>
<td></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>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>option</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td></td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td></td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td></td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td></td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td></td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td></td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td></td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td></td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td></td>
<td>STRH (immediate)</td>
</tr>
</tbody>
</table>
### Data Processing -- Register

These instructions are under the top-level.

#### Data-processing (2 source)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0110</td>
<td>Data-processing (2 source)</td>
</tr>
<tr>
<td>1 0110</td>
<td>Data-processing (1 source)</td>
</tr>
<tr>
<td>0 0xxx</td>
<td>Logical (shifted register)</td>
</tr>
<tr>
<td>0 1xx0</td>
<td>Add/subtract (shifted register)</td>
</tr>
<tr>
<td>0 1xx1</td>
<td>Add/subtract (extended register)</td>
</tr>
<tr>
<td>1 0000 000000</td>
<td>Add/subtract (with carry)</td>
</tr>
<tr>
<td>1 0000 x000001</td>
<td>Rotate right into flags</td>
</tr>
<tr>
<td>1 0000 xx00010</td>
<td>Evaluate into flags</td>
</tr>
<tr>
<td>1 0010 xxxx00x</td>
<td>Conditional compare (register)</td>
</tr>
<tr>
<td>1 0010 xxxx10x</td>
<td>Conditional compare (immediate)</td>
</tr>
<tr>
<td>1 0100</td>
<td>Conditional select</td>
</tr>
<tr>
<td>1 1xxx</td>
<td>Data-processing (3 source)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
Top-level encodings for A64

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>S</td>
<td>opcode</td>
<td>011xxx</td>
</tr>
<tr>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00011x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>001101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00111x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00001x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>000010</td>
<td>UDIV — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>000011</td>
<td>SDIV — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001010x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001000</td>
<td>LSV — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001001</td>
<td>LSRV — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001010</td>
<td>ASRV — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001111</td>
<td>RORV — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010x11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010000</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010001</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32H</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010010</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32W</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010100</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010101</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010110</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010000</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010111</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010000</td>
<td>SUBPS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010001</td>
<td>UDIV — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>000011</td>
<td>SDIV — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001001</td>
<td>IRG</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>000101</td>
<td>GMI</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010000</td>
<td>LSV — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010001</td>
<td>LSRV — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010101</td>
<td>ASRV — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010111</td>
<td>RORV — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001100</td>
<td>PACGA</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010xx0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010x0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010011</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010111</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>SUBPS</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>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>opcode2 opcode</td>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xxx1x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx1xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 00000 00011x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 00000 001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 00000 01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 00001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 00000 00000</td>
<td>RBIT — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 00000 00001</td>
<td>REV16 — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 00000 00010</td>
<td>REV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 00000 00011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 00000 000100</td>
<td>CLZ — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 00000 000101</td>
<td>CLS — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00000 00000</td>
<td>RBIT — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00000 00001</td>
<td>REV16 — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00000 00010</td>
<td>REV32</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00000 00011</td>
<td>REV — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00000 000100</td>
<td>CLZ — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00000 000101</td>
<td>CLS — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 00000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 00001</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 00010</td>
<td>PACDA, PACDZA — PACDA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 00011</td>
<td>PACDB, PACDZB — PACDB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 000100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 000101</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 000110</td>
<td>AUTDA, AUTDZA — AUTDA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 000111</td>
<td>AUTDB, AUTDBZ — AUTDB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001001</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001010</td>
<td>PACDA, PACDZA — PACDA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001011</td>
<td>PACDB, PACDZB — PACDB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001101</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001110</td>
<td>AUTDA, AUTDZA — AUTDA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001111</td>
<td>AUTDB, AUTDBZ — AUTDB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001000</td>
<td>XPACD, XPACI, XPACLRI — XPACI</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0001001</td>
<td>XPACD, XPACI, XPACLRI — XPACD</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 010000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 00001 0101xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Logical (shifted register)

These instructions are under [Data Processing -- Register](#).

| sf | S | Decode fields | opcode2 | opcode | Rn | Instruction Details | Architecture Version |
|----|---|---------------|---------|--------|---------------------|----------------------|
| 1  | 0 | 00001         | 011xxx  |        | UNALLOCATED         | -                    |

#### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>N</th>
<th>shift</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td></td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td></td>
<td></td>
<td>BIC (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td></td>
<td></td>
<td>ORR (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1</td>
<td></td>
<td></td>
<td>ORN (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
<td>EOR (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td>EON (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td>ANDS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td>BICS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td></td>
<td></td>
<td>AND (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td></td>
<td></td>
<td>BIC (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td></td>
<td></td>
<td>ORR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td></td>
<td></td>
<td>ORN (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
<td>EOR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td>EON (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td>ANDS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td>BICS (shifted register) — 64-bit</td>
</tr>
</tbody>
</table>

### Add/subtract (shifted register)

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>S</th>
<th>shift</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>S</th>
<th>shift</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td></td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td></td>
<td></td>
<td>ADD (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
<td>ADDS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td>SUB (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td></td>
<td></td>
<td>SUBS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td></td>
<td></td>
<td>ADD (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
<td>ADDS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td>SUB (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td>SUBS (shifted register) — 64-bit</td>
</tr>
</tbody>
</table>
Add/subtract (extended register)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>fields</th>
<th>opt</th>
<th>imm3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>11x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD (extended register) — 32-bit</td>
</tr>
<tr>
<td>0 0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADDS (extended register) — 32-bit</td>
</tr>
<tr>
<td>0 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SUB (extended register) — 32-bit</td>
</tr>
<tr>
<td>0 1 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SUBS (extended register) — 32-bit</td>
</tr>
<tr>
<td>1 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD (extended register) — 64-bit</td>
</tr>
<tr>
<td>1 0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADDS (extended register) — 64-bit</td>
</tr>
<tr>
<td>1 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SUB (extended register) — 64-bit</td>
</tr>
<tr>
<td>1 1 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SUBS (extended register) — 64-bit</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>fields</th>
<th>opt</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADC — 32-bit</td>
</tr>
<tr>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADCS — 32-bit</td>
</tr>
<tr>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SBC — 32-bit</td>
</tr>
<tr>
<td>0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SBCS — 32-bit</td>
</tr>
<tr>
<td>1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADC — 64-bit</td>
</tr>
<tr>
<td>1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADCS — 64-bit</td>
</tr>
<tr>
<td>1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SBC — 64-bit</td>
</tr>
<tr>
<td>1 1 1</td>
<td></td>
<td></td>
<td></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>fields</th>
<th>o2</th>
<th>imm6</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED -</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RMIF FEAT_FlagM</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED -</td>
</tr>
<tr>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED -</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>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>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>000000</td>
<td>0</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>SETF8, SETF16 -- SETF8</td>
<td>FEAT_FlagM</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>1</td>
<td>0</td>
<td>1101</td>
<td>SETF8, SETF16 -- SETF16</td>
<td>FEAT_FlagM</td>
</tr>
<tr>
<td>0</td>
<td>1</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>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>opcode2</th>
<th>o2</th>
<th>o3</th>
<th>cond</th>
<th>imm5</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>0</td>
<td></td>
<td></td>
<td></td>
<td>CCMN (register) -- 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>CCMP (register) -- 32-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>CCMN (register) -- 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>CCMP (register) -- 64-bit</td>
<td></td>
</tr>
</tbody>
</table>

**Conditional compare (immediate)**

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>opcode2</th>
<th>o2</th>
<th>o3</th>
<th>imm5</th>
<th>cond</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>0</td>
<td></td>
<td></td>
<td></td>
<td>CCMN (immediate) -- 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>CCMP (immediate) -- 32-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>CCMN (immediate) -- 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>CCMP (immediate) -- 64-bit</td>
<td></td>
</tr>
</tbody>
</table>
## Conditional select

These instructions are under **Data Processing -- Register**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>op2</th>
<th>Rm</th>
<th>cond</th>
<th>op2</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CSEL — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CSINC — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CSINV — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>CSNEG — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CSEL — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CSINC — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CSINV — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>CSNEG — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></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>op31</th>
<th>Rm</th>
<th>o0</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td>010</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td>011</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td>100</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td>110</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td>111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>MADD — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>1</td>
<td>MSUB — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>MADD — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>1</td>
<td>MSUB — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>0</td>
<td>SMADDL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>1</td>
<td>SMSUBL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>0</td>
<td>SMULH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>UMASSL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>1</td>
<td>UMSUBL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>UMLH</td>
<td></td>
<td></td>
<td></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>
<th>Architecture version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>Cryptographic AES</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx00</td>
<td>Cryptographic three-register SHA</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>Cryptographic two-register SHA</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>00</td>
<td>00xx</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD scalar copy</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>01</td>
<td>00xx</td>
<td>xxxxxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>0111</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD scalar three same FP16</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxxxxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>1111</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous FP16</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD scalar three same extra</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x100</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x110</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar pairwise</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>1xxxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>x1xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxxx00</td>
<td>Advanced SIMD scalar three different</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD scalar three same</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>10</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD scalar shift by immediate</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>11</td>
<td>xxxxxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>1x</td>
<td>xxxxxxxx0</td>
<td>Advanced SIMD scalar x indexed element</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx00</td>
<td>Advanced SIMD table lookup</td>
<td>-</td>
</tr>
<tr>
<td>0x00</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx10</td>
<td>Advanced SIMD permute</td>
<td>-</td>
</tr>
<tr>
<td>0x10</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx00</td>
<td>Advanced SIMD extract</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>00</td>
<td>00xx</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD copy</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>01</td>
<td>00xx</td>
<td>xxxxxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>0111</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>10xx</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD three same (FP16)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>10xx</td>
<td>xxxxxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>1111</td>
<td>00xxxxx10</td>
<td>Advanced SIMD two-register miscellaneous (FP16)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD three-register extension</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x100</td>
<td>00xxxxx10</td>
<td>Advanced SIMD two-register miscellaneous</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x110</td>
<td>00xxxxx10</td>
<td>Advanced SIMD across lanes</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x1xx</td>
<td>1xxxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>Address</td>
<td>Decode fields</td>
<td>Instruction Details</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>-----------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xx0 0x</td>
<td>x1xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xx0 0x</td>
<td>x1xx</td>
<td>Advanced SIMD three different</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xx0 0x</td>
<td>x1xx</td>
<td>Advanced SIMD three same</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xx0 10</td>
<td>0000</td>
<td>Advanced SIMD modified immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xx0 10</td>
<td>!= 0000</td>
<td>Advanced SIMD shift by immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xx0 11</td>
<td>xxxxxxxx1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xx0 1x</td>
<td>xxxxxxx00</td>
<td>Advanced SIMD vector x indexed element</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 00</td>
<td>10xx</td>
<td>Cryptographic three-register, imm2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 00</td>
<td>11xx</td>
<td>Cryptographic three-register SHA 512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 00</td>
<td>xxx0xxxxx</td>
<td>Cryptographic four-register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 01</td>
<td>00xx</td>
<td>XAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 01</td>
<td>1000</td>
<td>Cryptographic two-register SHA 512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xx0 1x</td>
<td>x0xx</td>
<td>Conversion between floating-point and fixed-point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0x1 0x</td>
<td>x1xx</td>
<td>Conversion between floating-point and integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0x1 0x</td>
<td>x1xx</td>
<td>Floating-point data-processing (1 source)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0x1 0x</td>
<td>x1xx</td>
<td>Floating-point data-processing (2 source)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0x1 0x</td>
<td>x1xx</td>
<td>Floating-point data-processing (3 source)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0x1 0x</td>
<td>x1xx</td>
<td>Floating-point conditional compare</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0x1 0x</td>
<td>x1xx</td>
<td>Floating-point conditional select</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0x1 1x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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](#).
## Cryptographic two-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>000</td>
<td>SHA1C</td>
</tr>
<tr>
<td>00</td>
<td>001</td>
<td>SHA1P</td>
</tr>
<tr>
<td>00</td>
<td>010</td>
<td>SHA1M</td>
</tr>
<tr>
<td>00</td>
<td>011</td>
<td>SHA1SU0</td>
</tr>
<tr>
<td>00</td>
<td>100</td>
<td>SHA256H</td>
</tr>
<tr>
<td>00</td>
<td>101</td>
<td>SHA256H2</td>
</tr>
<tr>
<td>00</td>
<td>110</td>
<td>SHA256SU1</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

## Advanced SIMD scalar copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1xxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>x1xxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>lxxxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00000</td>
<td>SHA1H</td>
</tr>
<tr>
<td>00</td>
<td>00001</td>
<td>SHA1SU1</td>
</tr>
<tr>
<td>00</td>
<td>00010</td>
<td>SHA256SU0</td>
</tr>
<tr>
<td>00</td>
<td>00011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

## Advanced SIMD scalar three same FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00000</td>
<td>DUP (element)</td>
</tr>
<tr>
<td>00</td>
<td>1xxx</td>
<td>UNALLOCATED</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>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>U a opcode</td>
<td>FMULX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 011</td>
<td>FCMEQ (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 0111</td>
<td>FRECPS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 100</td>
<td>FRSORTS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 011</td>
<td>FCMEQ (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 101</td>
<td>FACGE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 111</td>
<td>FACGT</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 010</td>
<td>FABD</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 100</td>
<td>FCMGT (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 101</td>
<td>FCMGT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 111</td>
<td>UNALLOCATED</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>U a opcode</td>
<td>FCVTNS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 1101</td>
<td>FCVTMS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 1110</td>
<td>SCVTFS (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 0110</td>
<td>FCMGT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 0111</td>
<td>FCMEQ (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 1110</td>
<td>FCMLT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11010</td>
<td>FCVTPS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11011</td>
<td>FCVTZS (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11101</td>
<td>FRECPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11111</td>
<td>FRECPX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11010</td>
<td>FCVTNU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11011</td>
<td>FCVTMU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U</td>
<td>a</td>
<td>opcode</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11010</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11011</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
</tr>
</tbody>
</table>

**Advanced SIMD scalar three same extra**

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>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 1 1 1 0</td>
<td>size</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>SQRDMLAH (vector)</td>
<td>FEAT_RDM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>SQRDMLSH (vector)</td>
<td>FEAT_RDM</td>
<td></td>
</tr>
</tbody>
</table>

**Advanced SIMD scalar two-register miscellaneous**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x011xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x10110</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Decode fields size</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---</td>
<td>-------------------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00011</td>
<td>SUQADD</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00111</td>
<td>SQABS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01000</td>
<td>CMGT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01001</td>
<td>CMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01010</td>
<td>CMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01011</td>
<td>ABS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10100</td>
<td>SOXTN, SOXTN2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11010</td>
<td>FCVTNS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11011</td>
<td>FCVTMS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11100</td>
<td>FCVTAS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11101</td>
<td>SCVT (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11110</td>
<td>FCMGT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>FCMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11112</td>
<td>FCMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11113</td>
<td>FCVTPS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11114</td>
<td>FCVTZS (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11115</td>
<td>FPRECPE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11116</td>
<td>FPRECPX</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00011</td>
<td>USQADD</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00111</td>
<td>SQNEG</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01000</td>
<td>CMGE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01001</td>
<td>CMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01011</td>
<td>NEG (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10010</td>
<td>SOXTN, SOXTN2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10100</td>
<td>UQXTN, UQXTN2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10110</td>
<td>FCVTNX, FCVTNX2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11010</td>
<td>FCVTVNU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11110</td>
<td>FCMGE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>FCMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11112</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11113</td>
<td>FCVTPU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11114</td>
<td>FCVTVZU (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11115</td>
<td>FRSORTE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11116</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**Advanced SIMD scalar pairwise**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
### Advanced SIMD scalar three different

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00xx</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>01xx</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>10xx</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>1100x</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>11010</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>111xx</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>1x 01101</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>11011</td>
<td>ADDP (scalar)</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>00 01100</td>
<td>FMAXNMP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 01101</td>
<td>FADD (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 01111</td>
<td>FMAXP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 01100</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>01 01101</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>01 01111</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>10 01100</td>
<td>FMINNMP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 01111</td>
<td>FMINP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 01100</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>11 01111</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>1 11011</td>
<td>UNALLOCATED</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>0x 01100</td>
<td>FMAXNMP (scalar) — single-precision and double-precision</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 01101</td>
<td>FADD (scalar) — single-precision and double-precision</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 01111</td>
<td>FMAXP (scalar) — single-precision and double-precision</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01100</td>
<td>FMINNMP (scalar) — single-precision and double-precision</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01111</td>
<td>FMINP (scalar) — single-precision and double-precision</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

---

### Decode fields

- **U**: Unavailable
- **size**: Unavailable
- **opcode**: Unavailable

### Instruction Details

- **ADDP (scalar)**
- **FMAXNMP (scalar)** — half-precision
- **FADD (scalar)** — half-precision
- **FMAXP (scalar)** — half-precision
- **FMINNMP (scalar)** — half-precision
- **FMINP (scalar)** — half-precision
- **FMINNMP (scalar)** — single-precision and double-precision
- **FMINP (scalar)** — single-precision and double-precision

---

Page 2411
### Advanced SIMD scalar three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>size</td>
<td>opcode</td>
</tr>
<tr>
<td>0 00000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 00100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 011xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 11011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 00001</td>
<td>SQADD</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 10000</td>
<td>ADD (vector)</td>
<td></td>
</tr>
<tr>
<td>0 10001</td>
<td>CMTST</td>
<td></td>
</tr>
<tr>
<td>0 10100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 10101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 10110</td>
<td>SQDMULH (vector)</td>
<td></td>
</tr>
<tr>
<td>0 10111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0x 11000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0x 11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0x 11010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0x 11011</td>
<td>FMULX</td>
<td></td>
</tr>
<tr>
<td>0 0x 11100</td>
<td>FCMEQ (register)</td>
<td></td>
</tr>
<tr>
<td>0 0x 11101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0x 11110</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0x 11111</td>
<td>FRECPS</td>
<td></td>
</tr>
<tr>
<td>0 1x 11000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1x 11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1x 11010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1x 11100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1x 11101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1x 11110</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1x 11111</td>
<td>FRSORTS</td>
<td></td>
</tr>
<tr>
<td>1 00001</td>
<td>UQADD</td>
<td></td>
</tr>
<tr>
<td>1 00101</td>
<td>UQSUB</td>
<td></td>
</tr>
<tr>
<td>1 00110</td>
<td>CMHI (register)</td>
<td></td>
</tr>
<tr>
<td>1 00111</td>
<td>CMHS (register)</td>
<td></td>
</tr>
<tr>
<td>1 01000</td>
<td>USHL</td>
<td></td>
</tr>
<tr>
<td>1 01001</td>
<td>UQSHL (register)</td>
<td></td>
</tr>
<tr>
<td>1 01010</td>
<td>URSHL</td>
<td></td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>01011</td>
<td>UQRSHL</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10000</td>
<td>SUB (vector)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10001</td>
<td>CMEQ (register)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10110</td>
<td>SQDRMULH (vector)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10111</td>
<td>UNALLOCATED</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>PABD</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.

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>immh</th>
<th>immb</th>
<th>opcode</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Advanced SIMD scalar shift by immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.
### Decode fields

<table>
<thead>
<tr>
<th>$U$</th>
<th>$!=$</th>
<th>0000</th>
<th>01000</th>
<th><strong>Instruction Details</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>01010</td>
<td><strong>S****H</strong>L</td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>01100</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>01110</td>
<td><strong>SQSHL</strong> (immediate)</td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>10000</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>10001</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>10010</td>
<td><strong>SQSHRN, SQSHRN2</strong></td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>10011</td>
<td><strong>SQRSRN, SQRSRN2</strong></td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>11100</td>
<td><strong>SCVT</strong> (vector, fixed-point)</td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>11110</td>
<td><strong>FCV</strong>TZS (vector, fixed-point)</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>00000</td>
<td><strong>US</strong>HR</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>00010</td>
<td><strong>US</strong>RA</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>00100</td>
<td><strong>UR</strong>SHR</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>00110</td>
<td><strong>UR</strong>SRA</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>01000</td>
<td><strong>SRI</strong></td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>01010</td>
<td><strong>SL</strong></td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>01100</td>
<td><strong>SQS</strong>HLU</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>01110</td>
<td><strong>USQ</strong>SHL (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>10000</td>
<td><strong>SQSHRUN, SQSHRUN2</strong></td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>10001</td>
<td><strong>SQRSRUN, SQRSRUN2</strong></td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>10010</td>
<td><strong>UQSHR, UQSHRN2</strong></td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>10011</td>
<td><strong>UORSHR, UQORSHRN2</strong></td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>11100</td>
<td><strong>UCVT</strong> (vector, fixed-point)</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>0000</td>
<td>11110</td>
<td><strong>FCV</strong>TZU (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](#).

Top-level encodings for A64
### Advanced SIMD table lookup

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

### Advanced SIMD permute

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
### Advanced SIMD extract

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<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>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>EXT</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</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></td>
<td>op imm5 imm4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>DUP (element)</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>DUP (general)</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>SMOV</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>UMOV</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0101</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x1000 0111</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>INS (element)</td>
</tr>
</tbody>
</table>

### Advanced SIMD extract

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
### Advanced SIMD two-register miscellaneous (FP16)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

#### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>a</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>FMAXNM (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>FMLA (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>FADD (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>FMULX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>FCMEQ (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>FMAX (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>111</td>
<td>FRECPSS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>000</td>
<td>FMINNM (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001</td>
<td>FMLS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>FSUB (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>110</td>
<td>FMIN (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>111</td>
<td>FRSQRTS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>FMAXNMP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>FADDP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>FMUL (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>FCMGE (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>FACGE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>FMAXP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>FDIV (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>000</td>
<td>FMINNMP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>FABD</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>FCMGT (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>FACGT</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>FMINP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Top-level encodings for A64

<table>
<thead>
<tr>
<th>Decode fields</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>U</td>
<td>a</td>
<td>opcode</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11000</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11001</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11010</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11011</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11101</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01100</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01101</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01110</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01111</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11001</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11010</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11011</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11100</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11110</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11001</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11010</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11011</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01111</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11001</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11010</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11011</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
</tr>
</tbody>
</table>

### Advanced SIMD three-register extension

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

**Advanced SIMD three-register extension**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
### Top-level encodings for A64

<table>
<thead>
<tr>
<th>Q</th>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000</td>
<td>SQRDMLAH (vector)</td>
<td>FEAT_RDM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>SQRDMLSH (vector)</td>
<td>FEAT_RDM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>UDOT (vector)</td>
<td>FEAT_DotProd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10xx</td>
<td>FCMLA</td>
<td>FEAT_FCMA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11x0</td>
<td>FCADD</td>
<td>FEAT_FCMA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1111</td>
<td>BFDOT (vector)</td>
<td>FEAT_BF16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1111</td>
<td>BFMLALB_BFMLAQT (vector)</td>
<td>FEAT_BF16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>011x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0100</td>
<td>SMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0101</td>
<td>USMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>1101</td>
<td>BFMMMLA</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0100</td>
<td>UMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0101</td>
<td>UNALLOCATED</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>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>011xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>11110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
<td>REV64</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>REV16 (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>SADDLP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00011</td>
<td>SUQADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>CLS (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00101</td>
<td>CNT</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>SADALP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>SQABS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>CMGT (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>CMEQ (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>CMLT (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>ABS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Decode fields size</td>
<td>opcode</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>----</td>
<td>-------------------</td>
<td>--------</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>0</td>
<td>10010</td>
<td>XTN, XTN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10011</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>SQXTN, SQXTN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 10110</td>
<td>FCVTN, FCVTN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 10111</td>
<td>FCVTL, FCVTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11000</td>
<td>PRINTN (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11001</td>
<td>PRINTM (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11010</td>
<td>FCVTNS (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11011</td>
<td>FCVTMS (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11100</td>
<td>FCVTAS (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11101</td>
<td>SCVTF (vector, integer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11110</td>
<td>PRINT32Z (vector)</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11111</td>
<td>PRINT64Z (vector)</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 01100</td>
<td>FCMGT (zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 01101</td>
<td>FCMEQ (zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 01110</td>
<td>FCMLT (zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 01111</td>
<td>FABS (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11000</td>
<td>PRINTP (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11001</td>
<td>PRINTZ (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11010</td>
<td>FCVTPS (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11011</td>
<td>FCVTZS (vector, integer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11100</td>
<td>URECPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11101</td>
<td>FRECPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 10110</td>
<td>BFCVTN, BFCVTN2</td>
<td>FEAT_BF16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>REV32 (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00010</td>
<td>UADLP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00011</td>
<td>USQADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00100</td>
<td>CLZ (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>UADALP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>SQNEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>CMGE (zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>CMLE (zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>NEG (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>SOXTUN, SOXTUN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td>SHLL, SHLL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UQXTN, UQXTN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 10110</td>
<td>FCVTXN, FCVTXN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 10111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11000</td>
<td>FRINTA (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11001</td>
<td>FRINTX (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11010</td>
<td>FCVTNU (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11011</td>
<td>FCVTMU (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11100</td>
<td>FCVTAU (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11101</td>
<td>UCVTF (vector, integer)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD across lanes

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x 11110</td>
<td>PRINT32X (vector)</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11111</td>
<td>PRINT64X (vector)</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00 00101</td>
<td>NOT</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01 00101</td>
<td>RBIT (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 00101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01100</td>
<td>FCMGE (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01101</td>
<td>FCMLE (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01111</td>
<td>FNEG (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11001</td>
<td>PRINTI (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11010</td>
<td>FCVTU (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11011</td>
<td>FCVTZU (vector, integer)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11100</td>
<td>URSORTE</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11101</td>
<td>FRSORTE</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11111</td>
<td>FSORT (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10 10110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

---

#### U [Q] | size | 1 1 0 0 0 | opcode | 1 0 | Rn | Rd
---

### Advanced SIMD across lanes

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0100x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1100x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>111xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00011</td>
<td>SADDLV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>SMAXV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11010</td>
<td>SMINV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11011</td>
<td>ADDV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 01100</td>
<td>FMAXNMV — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 01111</td>
<td>FMAXV — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 01100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 01111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 01100</td>
<td>FMINNMV — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 01111</td>
<td>FMINV — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 01100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 01111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
## Advanced SIMD three different

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

### Data Processing -- Scalar Floating-Point and Advanced SIMD

- **FMAXNMV**: single-precision and double-precision
- **FMAXV**: single-precision and double-precision
- **FMINNMV**: single-precision and double-precision
- **FMINV**: single-precision and double-precision

### Top-level encodings for A64

<table>
<thead>
<tr>
<th>Size</th>
<th>Opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00011</td>
<td>UADDLV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>UMAXV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11010</td>
<td>UMINV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>Size</th>
<th>Opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>01100</td>
<td>FMAXNMV</td>
</tr>
<tr>
<td>0x</td>
<td>01111</td>
<td>FMAXV</td>
</tr>
<tr>
<td>1x</td>
<td>01100</td>
<td>FMINNMV</td>
</tr>
<tr>
<td>1x</td>
<td>01111</td>
<td>FMINV</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SADDL, SADDL2</td>
</tr>
<tr>
<td>0001</td>
<td>SADDW, SADDW2</td>
</tr>
<tr>
<td>0010</td>
<td>SSUBL, SSUBL2</td>
</tr>
<tr>
<td>0011</td>
<td>SSUBW, SSUBW2</td>
</tr>
<tr>
<td>0100</td>
<td>ADDHN, ADDHN2</td>
</tr>
<tr>
<td>0101</td>
<td>SABAL, SABAL2</td>
</tr>
<tr>
<td>0110</td>
<td>SUBHN, SUBHN2</td>
</tr>
<tr>
<td>0111</td>
<td>SABDL, SABDL2</td>
</tr>
<tr>
<td>1000</td>
<td>SMLAL, SMLAL2 (vector)</td>
</tr>
<tr>
<td>1001</td>
<td>SODMLAL, SODMLAL2 (vector)</td>
</tr>
<tr>
<td>1010</td>
<td>SMLSL, SMLSL2 (vector)</td>
</tr>
<tr>
<td>1011</td>
<td>SODMLSL, SODMLSL2 (vector)</td>
</tr>
<tr>
<td>1100</td>
<td>SMULL, SMULL2 (vector)</td>
</tr>
<tr>
<td>1101</td>
<td>SODMULL, SODMULL2 (vector)</td>
</tr>
<tr>
<td>1110</td>
<td>PMULL, PMULL2</td>
</tr>
<tr>
<td>1000</td>
<td>UADDL, UADDL2</td>
</tr>
<tr>
<td>1001</td>
<td>UADDW, UADDW2</td>
</tr>
<tr>
<td>1010</td>
<td>USUBL, USUBL2</td>
</tr>
<tr>
<td>1011</td>
<td>USUBW, USUBW2</td>
</tr>
<tr>
<td>1010</td>
<td>RADDHN, RADDHN2</td>
</tr>
<tr>
<td>1011</td>
<td>UABAL, UABAL2</td>
</tr>
<tr>
<td>1100</td>
<td>UABDL, UABDL2</td>
</tr>
<tr>
<td>1101</td>
<td>UMULL, UMULL2 (vector)</td>
</tr>
<tr>
<td>1110</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Page 2422
### Advanced SIMD three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>SHADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>SQADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>SRHADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>SHSUB</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00101</td>
<td>SQSUB</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>CMGT (register)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>CMGE (register)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>SSHL</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>SQSHL (register)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>SRSRL</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>SQRSRL</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01100</td>
<td>SMAX</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01101</td>
<td>SMIN</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01110</td>
<td>SABD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01111</td>
<td>SABA</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
<td>ADD (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10001</td>
<td>CMTST</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10010</td>
<td>MLA (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10011</td>
<td>MUL (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>SMAXP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10101</td>
<td>SMINP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10110</td>
<td>SQDMULH (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10111</td>
<td>ADDP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11000</td>
<td>FMAXNM (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11001</td>
<td>FMLA (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11010</td>
<td>FADD (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11011</td>
<td>FMULX</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11100</td>
<td>FCMEQ (register)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11110</td>
<td>FMAX (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11111</td>
<td>FRECPS</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00011</td>
<td>AND (vector)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11101</td>
<td>FMLAL, FMLAL2 (vector) — FMLAL</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00011</td>
<td>BIC (vector, register)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11000</td>
<td>FMINNM (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11001</td>
<td>FMLS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11010</td>
<td>ESUB (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11011</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>11110</td>
<td>FMIN (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11111</td>
<td>FSQRTS</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>00011</td>
<td>ORR (vector, register)</td>
</tr>
</tbody>
</table>
## Top-level encodings for A64

<table>
<thead>
<tr>
<th>U Size</th>
<th>Opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 10</td>
<td>11101</td>
<td>FMLSL, FMLSL2 (vector) — FMLSL</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0 11</td>
<td>00011</td>
<td>ORN (vector)</td>
<td></td>
</tr>
<tr>
<td>0 11</td>
<td>11101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 00000</td>
<td>UHADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0001</td>
<td>UQADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00010</td>
<td>URHADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00100</td>
<td>UHSUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00101</td>
<td>UQSUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00110</td>
<td>CMHI (register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00111</td>
<td>CMHS (register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01000</td>
<td>USHL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01001</td>
<td>UQSLH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01010</td>
<td>URSHL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01011</td>
<td>UQRSHL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01100</td>
<td>UMAX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01101</td>
<td>UMIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01110</td>
<td>UABD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01111</td>
<td>UABA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10000</td>
<td>SUB (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10001</td>
<td>CMEQ (register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10010</td>
<td>MLS (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10011</td>
<td>PMUL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10100</td>
<td>UMAXP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10101</td>
<td>UMINP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10110</td>
<td>SORDMULH (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>11000</td>
<td>FMAXNMP (vector)</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>11010</td>
<td>FADDP (vector)</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>11011</td>
<td>FMUL (vector)</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>11100</td>
<td>FCMGE (register)</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>11101</td>
<td>FACGE</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>11110</td>
<td>FMAXP (vector)</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>11111</td>
<td>FDIV (vector)</td>
<td></td>
</tr>
<tr>
<td>1 00</td>
<td>00011</td>
<td>EOR (vector)</td>
<td></td>
</tr>
<tr>
<td>1 00</td>
<td>11001</td>
<td>FMLAL, FMLAL2 (vector) — FMLAL2</td>
<td></td>
</tr>
<tr>
<td>1 01</td>
<td>00011</td>
<td>BSL</td>
<td></td>
</tr>
<tr>
<td>1 01</td>
<td>11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 1x</td>
<td>11000</td>
<td>FMINNMP (vector)</td>
<td></td>
</tr>
<tr>
<td>1 1x</td>
<td>11010</td>
<td>FABD</td>
<td></td>
</tr>
<tr>
<td>1 1x</td>
<td>11011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 1x</td>
<td>11100</td>
<td>FCMGT (register)</td>
<td></td>
</tr>
<tr>
<td>1 1x</td>
<td>11101</td>
<td>FACGT</td>
<td></td>
</tr>
<tr>
<td>1 1x</td>
<td>11110</td>
<td>FMINP (vector)</td>
<td></td>
</tr>
<tr>
<td>1 1x</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 10</td>
<td>00011</td>
<td>BIT</td>
<td></td>
</tr>
<tr>
<td>1 10</td>
<td>11001</td>
<td>FMLSL, FMLSL2 (vector) — FMLSL2</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>1 11</td>
<td>00011</td>
<td>BIF</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD modified immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Q</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xx0</td>
<td>MOV — 32-bit shifted immediate</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0xx1</td>
<td>ORR (vector, immediate) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10x0</td>
<td>MOV — 16-bit shifted immediate</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10x1</td>
<td>ORR (vector, immediate) — 16-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>110x</td>
<td>MOV — 32-bit shifting ones</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>110x1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>MOV — 8-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11110</td>
<td>FMOV (vector, immediate) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>FMOV (vector, immediate) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>11111</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](#).

<table>
<thead>
<tr>
<th>Q</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xx0</td>
<td>MVNI — 32-bit shifted immediate</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0xx1</td>
<td>BIC (vector, immediate) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10x0</td>
<td>MVNI — 16-bit shifted immediate</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10x1</td>
<td>BIC (vector, immediate) — 16-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>110x</td>
<td>MVNI — 32-bit shifting ones</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>MOV — 64-bit scalar</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>MOV — 64-bit vector</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>FMOV (vector, immediate) — double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: immh != 0000 & immh != 0000

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00001</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>0111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
## Top-level encodings for A64

### Advanced SIMD vector x indexed element

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

### Decode fields

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1011x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0  00000</td>
<td>SSRH</td>
</tr>
<tr>
<td>0  00010</td>
<td>SSRH</td>
</tr>
<tr>
<td>0  00100</td>
<td>SSRH</td>
</tr>
<tr>
<td>0  00110</td>
<td>SSRH</td>
</tr>
<tr>
<td>0  01000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0  01010</td>
<td>SHL</td>
</tr>
<tr>
<td>0  01100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0  01110</td>
<td>SQRSHL (immediate)</td>
</tr>
<tr>
<td>0  10000</td>
<td>SHRN, SHRN2</td>
</tr>
<tr>
<td>0  10001</td>
<td>RSHRN, RSHRN2</td>
</tr>
<tr>
<td>0  10010</td>
<td>SQRSHRN, SQRSHRN2</td>
</tr>
<tr>
<td>0  10011</td>
<td>SQRSHRN, SQRSHRN2</td>
</tr>
<tr>
<td>0  10100</td>
<td>SSHLL, SSHLL2</td>
</tr>
<tr>
<td>0  11100</td>
<td>SCVT (vector, fixed-point)</td>
</tr>
<tr>
<td>0  11111</td>
<td>FCVTZ (vector, fixed-point)</td>
</tr>
<tr>
<td>1  00000</td>
<td>USHR</td>
</tr>
<tr>
<td>1  00010</td>
<td>USRHR</td>
</tr>
<tr>
<td>1  00110</td>
<td>USRHR</td>
</tr>
<tr>
<td>1  01000</td>
<td>SRI</td>
</tr>
<tr>
<td>1  01010</td>
<td>SLI</td>
</tr>
<tr>
<td>1  01100</td>
<td>SQRSHL</td>
</tr>
<tr>
<td>1  01110</td>
<td>UQSHL (immediate)</td>
</tr>
<tr>
<td>1  10000</td>
<td>SQSHRUN, SQSHRUN</td>
</tr>
<tr>
<td>1  10001</td>
<td>SQSHRUN, SQSHRUN</td>
</tr>
<tr>
<td>1  10010</td>
<td>SQSHRUN, SQSHRUN</td>
</tr>
<tr>
<td>1  10011</td>
<td>SQSHRUN, SQSHRUN</td>
</tr>
<tr>
<td>1  10100</td>
<td>SSHLL, SSHLL2</td>
</tr>
<tr>
<td>1  11100</td>
<td>UCVTF (vector, fixed-point)</td>
</tr>
<tr>
<td>1  11111</td>
<td>FCVTU (vector, fixed-point)</td>
</tr>
</tbody>
</table>

### Architecture Version

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>011001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>U size</td>
<td>U opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>SMLAL, SMLAL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>SQDMLAL, SQDMLAL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>SMLSL, SMLSL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>SQDMLSL, SQDMLSL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>MUL (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>SMULL, SMULL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>SQDMLU, SQDMLU2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td>SQDMULH (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>SQRDMULH (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>SDOT (by element)</td>
</tr>
<tr>
<td>0 x</td>
<td>0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 x</td>
<td>0100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>FMLA (by element)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>FMLS (by element)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>FMUL (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>SUDOT (by element)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>BFDOT (by element)</td>
</tr>
<tr>
<td>1 x</td>
<td>0001</td>
<td>FMLA (by element)</td>
</tr>
<tr>
<td>1 x</td>
<td>0101</td>
<td>FMLS (by element)</td>
</tr>
<tr>
<td>1 x</td>
<td>1001</td>
<td>FMUL (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>FMLAL, FMLAL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>FMLSL, FMLSL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>USDOT (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>BFMLALB, BFMLALT (by element)</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>MLA (by element)</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>UMLAL, UMLAL2 (by element)</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>MLS (by element)</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>UMLSL, UMLSL2 (by element)</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>UMULL, UMULL2 (by element)</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>SQRDMLAH (by element)</td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>UDOT (by element)</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>SQRDMLSH (by element)</td>
</tr>
<tr>
<td>1 x</td>
<td>1000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 x</td>
<td>1100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>FMULX (by element)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>FCMLA (by element)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>FMULX (by element)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>FCMLA (by element)</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>00</td>
<td>SM3TT1A</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>01</td>
<td>SM3TT1B</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>10</td>
<td>SM3TT2A</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>11</td>
<td>SM3TT2B</td>
<td>FEAT SM3</td>
</tr>
</tbody>
</table>

Cryptographic 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>SHA512H</td>
<td>FEAT SHA512</td>
</tr>
<tr>
<td>01</td>
<td>SHA512H2</td>
<td>FEAT SHA512</td>
</tr>
<tr>
<td>10</td>
<td>SHA512SU1</td>
<td>FEAT SHA512</td>
</tr>
<tr>
<td>11</td>
<td>RAX1</td>
<td>FEAT SHA3</td>
</tr>
<tr>
<td>00</td>
<td>SM3PARTW1</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>01</td>
<td>SM3PARTW2</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>10</td>
<td>SM4EKEY</td>
<td>FEAT SM4</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Cryptographic four-register

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FOR3</td>
<td>FEAT SHA3</td>
</tr>
<tr>
<td>01</td>
<td>BCAX</td>
<td>FEAT SHA3</td>
</tr>
</tbody>
</table>
### Cryptographic two-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>SM3SS1</td>
<td>FEAT_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</th>
<th>S</th>
<th>ptype</th>
<th>Decode fields</th>
<th>opcode</th>
<th>scale</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x0</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x</td>
<td>00x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1x</td>
<td>00x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1x</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0xxx</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td></td>
<td>00</td>
<td>010</td>
<td>SCVT (scalar, fixed-point) — 32-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td></td>
<td>00</td>
<td>011</td>
<td>UCVT (scalar, fixed-point) — 32-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td></td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, fixed-point) — single-precision to 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td></td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, fixed-point) — single-precision to 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td></td>
<td>00</td>
<td>010</td>
<td>SCVT (scalar, fixed-point) — 32-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td></td>
<td>00</td>
<td>011</td>
<td>UCVT (scalar, fixed-point) — 32-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td></td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, fixed-point) — double-precision to 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td></td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, fixed-point) — double-precision to 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td></td>
<td>00</td>
<td>010</td>
<td>SCVT (scalar, fixed-point) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td></td>
<td>00</td>
<td>011</td>
<td>UCVT (scalar, fixed-point) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>
### Instruction Details

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>scale</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 64-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 64-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 64-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 64-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 64-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 64-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>

### Conversion between floating-point and integer

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

#### Top-level encodings for A64

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>FCVTNS (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>FCVTNU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>SCVTF (scalar, integer) — 32-bit to single-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>UCVTF (scalar, integer) — 32-bit to single-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>FCVTAS (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>FCVTAU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>Decode fields</td>
<td>ptype</td>
<td>rmode</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>---------------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 110</td>
<td>FMOV (general) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 111</td>
<td>FMOV (general) — 32-bit to single-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 01 000</td>
<td>FCVT (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 01 001</td>
<td>FCVT (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1x 11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 10 000</td>
<td>FCVT (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 10 001</td>
<td>FCVT (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 11 000</td>
<td>FCVT (scalar, integer) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 11 001</td>
<td>FCVT (scalar, integer) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0x 11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 000</td>
<td>FCVT (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 001</td>
<td>FCVT (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 010</td>
<td>SCVT (scalar, integer) — 32-bit to double-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 011</td>
<td>UCVTF (scalar, integer) — 32-bit to double-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 01 100</td>
<td>FCVTAS (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 01 101</td>
<td>FCVTAU (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 01 000</td>
<td>FCVT (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 01 001</td>
<td>FCVT (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 000</td>
<td>FCVT (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 001</td>
<td>FCVT (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 000</td>
<td>FCVT (scalar, integer) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 001</td>
<td>FCVT (scalar, integer) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 110</td>
<td>FICVT (scalar, integer) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10 11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 000</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 001</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 010</td>
<td>SCVT (scalar, integer) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 011</td>
<td>UCVTF (scalar, integer) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 100</td>
<td>FCVTAS (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 10 110</td>
<td>FMOV (general) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 01 000</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 01 001</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 10 000</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 10 001</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 11 000</td>
<td>FCVT (scalar, integer) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 11 001</td>
<td>FCVT (scalar, integer) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>Decode fields</td>
<td>ptype</td>
<td>rmode</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>--------------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00</td>
<td>000</td>
<td></td>
<td>FCVTNS (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00</td>
<td>001</td>
<td></td>
<td>FCVNU (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, integer) — 64-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, integer) — 64-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00</td>
<td>100</td>
<td></td>
<td>FCVTAS (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00</td>
<td>101</td>
<td></td>
<td>FCVTAU (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 01</td>
<td>000</td>
<td></td>
<td>FCVTPS (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 01</td>
<td>001</td>
<td></td>
<td>FCVTPU (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 10 00</td>
<td>000</td>
<td></td>
<td>FCVTMS (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 10 00</td>
<td>001</td>
<td></td>
<td>FCVTMU (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 11 00</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, integer) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 11 00</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, integer) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 x1 11x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 00</td>
<td>000</td>
<td></td>
<td>FCVTNS (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 00</td>
<td>001</td>
<td></td>
<td>FCVNU (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 01</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, integer) — 64-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 01</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, integer) — 64-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 10 00</td>
<td>000</td>
<td></td>
<td>FCVTMS (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 10 00</td>
<td>001</td>
<td></td>
<td>FCVTMU (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 11 00</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, integer) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 11 00</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, integer) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 x0 11x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 01 11x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 01 111</td>
<td></td>
<td></td>
<td>FMOV (general) — top half of 128-bit to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 01 111</td>
<td></td>
<td></td>
<td>FMOV (general) — 64-bit to top half of 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 1x 11x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 00</td>
<td>000</td>
<td></td>
<td>FCVTNS (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 00</td>
<td>001</td>
<td></td>
<td>FCVNU (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 01</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, integer) — 64-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 01</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, integer) — 64-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 00</td>
<td>000</td>
<td></td>
<td>FCVTNS (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 00</td>
<td>001</td>
<td></td>
<td>FCVNU (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 10</td>
<td>010</td>
<td></td>
<td>FMOV (general) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 11</td>
<td>011</td>
<td></td>
<td>FMOV (general) — 64-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 00</td>
<td>000</td>
<td></td>
<td>FCVTNS (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>
Floating-point data-processing (1 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.
### Top-level encodings for A64

#### Floating-point compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

### Decode fields

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001000</td>
<td>FRINTN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001001</td>
<td>FRINTP (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001010</td>
<td>FRINTM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001011</td>
<td>FRINTZ (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001100</td>
<td>FRINTA (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001110</td>
<td>FRINTX (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001111</td>
<td>FRINTI (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010000</td>
<td>FRINT32Z (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010001</td>
<td>FRINT32X (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010010</td>
<td>FRINT64Z (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010011</td>
<td>FRINT64X (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0101xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000000</td>
<td>FMOV (register) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000001</td>
<td>FABS (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000010</td>
<td>FNEG (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000011</td>
<td>FSQRT (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000100</td>
<td>FCVT — half-precision to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000101</td>
<td>FCVT — half-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001111x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>010000</td>
<td>FRINTN (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>010001</td>
<td>FRINTP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>010100</td>
<td>FRINTM (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>010101</td>
<td>FRINTZ (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>011000</td>
<td>FRINTA (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>011001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>011100</td>
<td>FRINTX (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>011111</td>
<td>FRINTI (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

---

[Data Processing -- Scalar Floating-Point and Advanced SIMD](#)

---

### Floating-point compare

#### These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.
### Floating-point immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

![Floating-point immediate table](#)

### Floating-point conditional compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

![Floating-point conditional compare 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>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>FCCMP — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>FCCMPE — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

### Decode fields

<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>0000</td>
<td>FMUL (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0001</td>
<td>FDIV (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0010</td>
<td>FADD (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0011</td>
<td>FSUB (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0100</td>
<td>FMAX (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0101</td>
<td>FMIN (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0110</td>
<td>FMAXNM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0111</td>
<td>FMINNM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1000</td>
<td>FNMUL (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0000</td>
<td>FMUL (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0001</td>
<td>FDIV (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0010</td>
<td>FADD (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0011</td>
<td>FSUB (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0100</td>
<td>FMAX (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0101</td>
<td>FMIN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0110</td>
<td>FMAXNM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0111</td>
<td>FMINNM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1000</td>
<td>FNMUL (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>0000</td>
<td>FMUL (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>0001</td>
<td>FDIV (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>0010</td>
<td>FADD (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>0011</td>
<td>FSUB (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>0100</td>
<td>FMAX (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>0101</td>
<td>FMIN (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>0110</td>
<td>FMAXNM (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>0111</td>
<td>FMINNM (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></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>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 00</td>
<td>FCSEL — single-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 01</td>
<td>FCSEL — double-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 11</td>
<td>FCSEL — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></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>o1</th>
<th>o0</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 00 0 0</td>
<td>FMADD — single-precision</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 00 0 1</td>
<td>FMSUB — single-precision</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 00 1 0</td>
<td>FNMADD — single-precision</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 01 0 1</td>
<td>FMSUB — double-precision</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 11 0 1</td>
<td>FNMSUB — double-precision</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 11 0 0</td>
<td>FMADD — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 11 0 1</td>
<td>FMSUB — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 11 1 0</td>
<td>FNMADD — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 11 1 1</td>
<td>FNMSUB — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages.

Pseudocodes

Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

// AArch32.VCRMatch()  // ================

boolean AArch32.VCRMatch(bits(32) vaddress)

    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);
        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && !IsSecure() then
                match_word<UInt(vaddress<4:2>) + 24> = '1';  // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';  // Secure vectors (or no EL3)
        endif
        if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
            match_word<UInt(vaddress<4:2>) + 8> = '1';  // Monitor vectors
        endif
        // Mask out bits not corresponding to vectors.
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110';  // DBGVCR[31:8] are RES0
        elif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110';  // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';
        endif
        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 = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
        endif
    else
        match = FALSE;
    endif
    return match;

Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()  // ================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()

    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());
assert n <= UInt(DBGDIR.BRPs);

enabled = 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 = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
if value_mismatch && !mismatch_i then
  value_mismatch = ConstrainUnpredictableBool(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 DBGVR[n]+2.
  if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
  if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

match = value_match && state_match && enabled;
mismatch = value_mismatch && state_match && enabled;

return (match, mismatch);
Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointValueMatch
// 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 n > UInt(DBGDIDR.BRPs) then        // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs), Unpredictable_BPNOTIMPL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);
endif

context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
// Determine what to compare against.
match_addr = (dbgtype == '0x0x');
mismatch = (dbgtype == '010x');
match_vmid = (dbgtype == '10xx');
m_match_cid1 = (dbgtype == 'xx1x');
m_match_cid2 = (dbgtype == '11xx');
linked = (dbgtype == 'xxx1');

if linked_to && (!linked || match_addr) then return (FALSE,FALSE);
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');
    integer top = 31;
    BVR_match = (vaddress<top:2> == DBGBVR[n]<top:2>) && byte_select_match;
elif m_match_cid1 then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
elif !HaveVirtHostExt() && !HaveV82Debug() then
    vmid = ZeroExtend(VTTBR.VMID, 16);
    bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
elif !Have16bitVMID() || VTCR_EL2.VS == '0' then
    vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
endif
if !linked_to then return (FALSE,FALSE);
if context_aware then
    if mismatch then
        assert vmid IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if vmid == Constraint_DISABLED then return (FALSE,FALSE);
    endif
endif

// 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);

// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return (FALSE,FALSE);

// If called 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 then return (FALSE,FALSE);
if context_aware && match_addr then return (FALSE,FALSE);

// Shared Pseudocode Functions
Page 2441
bvr_vmid = \texttt{ZeroExtend}(\texttt{DBGBXVR}[n]<7:0>, 16);
else
  vmid = \texttt{VTBR\_EL2\_VMID};
bvr_vmid = \texttt{DBGBXVR}[n]<15:0>;
BXVR\_match = (\texttt{PSTATE\_EL IN (EL0, EL1)} \&\& \texttt{EL2Enabled()} \&\&
  vmid == bvr_vmid);
elsif match\_cid2 then
  BXVR\_match = ((\texttt{HaveVirtHostExt()} || \texttt{HaveV82Debug()} \&\& \texttt{EL2Enabled()} \&\&
  \texttt{!ELUsingAArch32(EL2)} \&\&
  \texttt{DBGBXVR}[n]<31:0> == \texttt{CONTEXTIDR\_EL2}<31:0>);

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 \&\& \texttt{!mismatch}, \texttt{!match \&\& mismatch});
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;
    elseif 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
    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 CONRAINED 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
            end case
        end if
    end if

    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

```java
// AArch32.GenerateDebugExceptionsFrom()
// -------------------------------------

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)
    if from == EL0 && (ELIsInHost(from) || !ELStateUsingAArch32(EL1, secure)) then
        mask = bit UNKNOWN; // PSTATE.D mask, unused for EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);
    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    if HaveEL(EL3) && secure then
        assert from != EL2; // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3.SDD == '0';
        else
            spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
            if spd<1> == '1' then
                enabled = spd<0> == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
            else
                enabled = from != EL2;
        else
            enabled = from != EL2;
    return enabled;
```

Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```java
// AArch32.CheckForPMUOverflow()
// =============================

// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch32.CheckForPMUOverflow()
    if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();
    pmuirq = PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSSET<31> == '1';
    for n = 0 to UINT(PMCR.N) - 1
        if HaveEL(EL2) then
            hpnm = if ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
            hpme = if ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
            E = (if n < UINT(hpnm) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSSET<n> == '1' then pmuirq = TRUE;
    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)
    return pmuirq;
```
Library pseudocode for aarch32/debug/pmu/AArch32.CountEvents
boolean AArch32.CountEvents(integer n)
assert n == 31 || n < UInt(PMCR.N);
if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);
// Event counting is disabled in Debug state
debug = Halted();

if !ELUsingAArch32(EL2) then
    hpmm = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
    hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
    resvd_for_el2 = n >= UInt(hpmm) && n != 31;
else
    resvd_for_el2 = FALSE;

// Main enable controls
if resvd_for_el2 then
    E = if !ELUsingAArch32(EL2) then HDCR.HPME else MDCR_EL2.HPME;
else
    E = PMCR.E;
enabled = E == '1' && PMCNTENSET<n> == '1';

// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;
// Event counting in Secure state is prohibited if all of:
// * EL3 is implemented
// * One of the following is true:
//   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
//     - FEAT_PMUv3p7 is not implemented
//     - MDCR_EL3.MPMX == 0
//   - EL3 is using AArch32 and SDCR.SPME == 0
//   - Not executing at EL0, or SDER.SUNIDEN == 0
if HaveEL(EL3) && IsSecure() then
    spme = if !ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
    if !ELUsingAArch32(EL3) && HavePMUv3p7() then
        prohibited = spme == '0' && MDCR_EL3.MPMX == '0';
    else
        prohibited = spme == '0';
    if prohibited && PSTATE.EL == EL0 then
        prohibited = SDER.SUNIDEN == '0';

if !prohibited && PSTATE.EL == EL2 then HDCR.HCCD == '1' then
    prohibited = TRUE;
// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * HDCR.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
    prohibited = HDCR.HPMD == '1';
// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebuggerEnabled();
// PMCR.DP disables the cycle counter when event counting is prohibited
if enabled && prohibited && n == 31 then
    enabled = PMCR.DP == '0';
// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR.DP.
if Havev85PMU() && n == 31 then
    if HaveEL(EL3) && IsSecure() then
        sccd = if !ELUsingAArch32(EL3) then SDCR.SCCD else MDCR_EL3.SCCD;
        if sccd == '1' then prohibited = TRUE;
    if PSTATE.EL == EL2 && HDCR.HCCD == '1' then
        prohibited = TRUE;
// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() && n != 31 then
    ovflw = PMOVSR<UInt(PMCR.N)-1:0>;
    if resvd_for_el2 then
        FZ = if ELUsingAArch32(EL2) then HDCR.HPMFZO else MDCR_EL2.HPMFZO;
        ovflw<UInt(hpmn)-1:0> = Zeros();
    else
        FZ = PMCR.FZO;
        if HaveEL(EL2) then
            ovflw<UInt(PMCR.N)-1:UInt(hpmn)> = Zeros();
        end
    end
    frozen = FZ == '1' && !IsZero(ovflw);
end

// Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];
P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
case PSTATE.EL of
    when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
    when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
    when EL2 filtered = NSH == '0';
    when EL3 filtered = P == '1';
end

return !debug && enabled && !prohibited && !filtered && !frozen;

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.IE = '0';
PSTATE.IT = '00000000';
if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
EDSCR.ERR = '1';
UpdateEDSCRFields();
EndOfInstruction();
// AArch32.EnterModeInDebugState()
// =======================================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    // Use the AArch32 entry point to map to the desired mode.
    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.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSR.ERR = '1';
    UpdateEDSCRFields();                      // Update EDSCR processor state flags.
EndOfInstruction();

// AArch32.EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    // Use the AArch32 entry point to map to the desired mode.
    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;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    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 = bit UNKNOWN;
        DLR = bits(32) UNKNOWN;
        DSPSR = bits(32) UNKNOWN;
        EDSR.ERR = '1';
        UpdateEDSCRFields();                      // Update EDSCR processor state flags.
EndOfInstruction();
boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

integer top = 31;
bottom = if DBGWVR[n]<2> == '1' then 2 else 3;       // Word or doubleword
byte_select_match = (DBGWCR[n].BAS<UInt>(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR[n].MASK);

// If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', 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));  // Not contiguous
  MSB = (DBGWCR[n].BAS + LSB);
  if !IsZero(MSB AND (MSB - 1)) then                    // Not contiguous
    byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBACONTIGUOUS);
  bottom = 3;                                       // For the whole doubleword

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then
  (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
  assert c IN {
    Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
  case c of
    when Constraint_DISABLED return FALSE;           // Disabled
    when Constraint_NONE mask = 0;                   // No masking
    // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
    if mask > bottom then
      // If the DBGxVR<n> EL1.RESS field bits are not a sign extension of the MSB
      // of DBGBV<n> EL1.VA, it is UNPREDICTABLE whether they appear to be
      // included in the match.
      if !IsOnes(DBGBV[n]<63:top>) && !IsZero(DBGBV[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
          top = 63;
          WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
          WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
      else
        WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;
      end if
    end if
  end case
return WVR_match && byte_select_match;
Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());
assert n <= UInt(DBGDIDR.WRPs);

// "ispriv" is:
//  * FALSE for all loads, stores, and atomic operations executed at EL0.
//  * FALSE if the access is unprivileged.
//  * TRUE for all other loads, stores, and atomic operations.

enabled = DBGWCR[n].E == '1';
linked = DBGWCR[n].WT == '1';
isbreakpnt = FALSE;

state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC, linked, DBGWCR[n].LBN, isbreakpnt, ispriv);
ls_match = FALSE;
ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)>) == '1';

value_match = FALSE;
for byte = 0 to size - 1
value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);
return value_match && state_match && ls_match && enabled;

Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) || (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) || IsDebugException(fault) && MDCR_EL2.TDE == '1');

if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

if route_to_aarch64 then
AArch64.Abort(ZeroExtend(vaddress), fault);
elsif fault.acctype == AccType_IFETCH then
AArch32.TakePrefetchAbortException(vaddress, fault);
else
AArch32.TakeDataAbortException(vaddress, fault);
Library pseudocode for aarch32/exceptions/aborts/AArch32.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()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

// The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor
// format. Normally, the current translation table format determines the format. For an abort
// from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if
// any of the following applies:
// * The Secure TTBCR.EAE is set to 1.
// * The abort is synchronous and either:
//   - It is taken from Hyp mode.
//   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.

long_format = FALSE;
if route_to_monitor && !IsSecure() then
    long_format = TTBCR_S.EAE == '1';
else
    long_format = TTBCR.EAE == '1';

if !IsSErrorInterrupt(fault) && !long_format then
    long_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
else
    long_format = TTBCR.EAE == '1';

if fault.acctype == AccType_IC then
    if (!long_format &&
        boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR")
        then
        i_syndrome = syndrome;
        syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
    else
        i_syndrome = bits(32) UNKNOWN;

if route_to_monitor then
    IFSR_S = i_syndrome;
    else
    IFSR = i_syndrome;

if route_to_monitor then
    DFSR_S = syndrome;
    DFAR_S = vaddress;
else
    DFSR = syndrome;
    DFAR = vaddress;

return;
// 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;

// 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) && HCR.FIQ == '1')));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;
    lr_offset = 4;
    if IsDebugException(fault) then DBGSCRext.MODE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif 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/asynch/AArch32.TakePhysicalFIQException

// 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' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';
    if route_to_aarch64 then
        AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                   (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
// AArch32.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.
AArch32.TakePhysicalIRQException()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.IRQ == '1';
if route_to_aarch64 then AArch64.TakePhysicalIRQException();

route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
    (HCR.TGE == '1' || HCR.IMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x18;
lr_offset = 4;
if route_to_monitor then
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
    exception = ExceptionSyndrome(Exception_IRQ);
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/asynch/AArch32.TakePhysicalSErrorException

// AArch32.TakePhysicalSErrorException()
// -------------------------------------

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) errortype, boolean impdef_syndrome, bits(24) full_syndrome)

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 & 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 = 0;
if IsSErrorEdgeTriggered(full_syndrome) then
  ClearPendingPhysicalSError();

fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
vaddress = bits(32) UNKNOWN;
if route_to_monitor then
  AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
  AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
  exception = AArch32.AbortSyndrome(Exc_DataAbort, fault, vaddress);
  if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
  else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
  else
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abt, 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(parity, errortype, extflag);

ClearPendingVirtualSError();
AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// ============================
AArch32.SoftwareBreakpoint(bits(16) immediate)
    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH;           // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;
    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '1010';
```

Library pseudocode for aarch32/exceptions/exceptions/
AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =======================================
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.
AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)
    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;
        if (tid0 == '1' && reg == '0000')                             // FPSID
            || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then    // MVFRx
                if ELUsingAArch32(EL2) then
                    AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
                else
                    AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
```
// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(int,bit) AArch32.ExceptionClass(Exception exctype)

  il_is_valid = TRUE;

case exctype of
    when Exception_Uncategorized ec = 0x00; il_is_valid = FALSE;
    when Exception_WFxTrap ec = 0x01;
    when Exception_CP15RTTrap ec = 0x03;
    when Exception_CP15RRTTrap ec = 0x04;
    when Exception_CP14RTTrap ec = 0x05;
    when Exception_CP14DTTtrap ec = 0x06;
    when Exception_AdvSIMDFPAccessTrap ec = 0x07;
    when Exception_FPIDTrap ec = 0x08;
    when Exception_PACTrap ec = 0x09;
    when Exception_LDST64BTrap ec = 0x0A;
    when Exception_CP14RTTrap ec = 0x0C;
    when Exception_IllegalState ec = 0x0E; il_is_valid = FALSE;
    when Exception_SupervisorCall ec = 0x11;
    when Exception_HypervisorCall ec = 0x12;
    when Exception_MonitorCall ec = 0x13;
    when Exception_InstructionAbort ec = 0x20; il_is_valid = FALSE;
    when Exception_PCAlignment ec = 0x22; il_is_valid = FALSE;
    when Exception_DataAbort ec = 0x24;
    when Exception_NV2DataAbort ec = 0x25;
    when Exception_FPTrappedException ec = 0x28;
    otherwise unreachable();

  if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
    ec = ec + 1;

  if il_is_valid then
    il = if ThisInstrLength() == 32 then '1' else '0';
  else
    il = '1';

  return (ec,il);

Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()

  return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
          EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
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);
// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                     // Secure state
    elseif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set 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) then
        if MVBAR<0> == '1' then       // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    BranchTo(rv, BranchType_RESET);

Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

// AArch32.FPTrappedException()
// ===============

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
   if AArch32.GeneralExceptionsToAArch64() then
      is_ase = FALSE;
      element = 0;
      AArch64.FPTrappedException(is_ase, accumulated_exceptions);
      FPEXC.DEX = '1';
      FPEXC.TFV = '1';
      FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;
                      // IDF,IXF,UFF,OFF,DZF,IOF
      FPEXC<10:8>  = '111';  // VECITR is RES1
   AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

// AArch32.CallHypervisor()
// ===============

// Performs a HVC call
AArch32.CallHypervisor(bits(16) immediate)
   assert HaveEL(EL2);
   if !ELUsingAArch32(EL2) then
      AArch64.CallHypervisor(immediate);
   else
      AArch32.TakeHVCException(immediate);

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

// AArch32.CallSupervisor()
// ===============

// Calls the Supervisor
AArch32.CallSupervisor(bits(16) immediate)
   if AArch32.CurrentCond() != '1110' then
      immediate = bits(16) UNKNOWN;
   if AArch32.GeneralExceptionsToAArch64() then
      AArch64.CallSupervisor(immediate);
   else
      AArch32.TakeSVCException(immediate);

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCException

// AArch32.TakeHVCException()
// ===============

AArch32.TakeHVCException(bits(16) immediate)
   assert HaveEL(EL2) & ELUsingAArch32(EL2);
   AArch32.ITAdvance();
   SSAdvance();
   bits(32) preferred_exception_return = NextInstrAddr();
   vect_offset = 0x08;
   exception = ExceptionSyndrome(Exception_HypervisorCall);
   exception.syndrome<15:0> = immediate;
   if PSTATE.EL == EL2 then
      AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
   else
      AArch32nEnterHypMode(exception, preferred_exception_return, 0x14);
// 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);

// 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);
Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return, integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if !((exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    endIf;
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTRL.TE;                   // PSTATE.J is RES0
    PSTATE.SS = '0';
    ShouldAdvanceSS = FALSE;
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTRL.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = HSCTRL.DSSBS;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
EndOfInstruction();

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                   // PSTATE.J is RES0
    PSTATE.SS = '0';
    ShouldAdvanceSS = FALSE;
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    endIf;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
EndOfInstruction();
// 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();
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    ShouldAdvanceSS = FALSE;
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.I = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
        if HaveSSBSExt() then
            PSTATE.SSBS = SCTLR.DSSBS;
            BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
    EndOfInstruction();
AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)
AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)
    if PSTATE.EL == EL0 && (!HaveEL(EL2) || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0') && !ELUsingAArch32(EL1)) && !ELUsingAArch32(EL0) then
        // The PE behaves as if FPEXC.EN is 1
        AArch64.CheckFPAdvSIMDEnabled();
    elsif PSTATE.EL == EL0 && HaveEL(EL2) && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && !ELUsingAArch32(EL1) then
        if fpexc_check && HCR_EL2.RW == '0' then
            fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
            if fpexc_en == '0' then UNDEFINED;
            AArch64.CheckFPAdvSIMDEnabled();
        else
            cpacr_asedis = CPACR.ASEDIS;
            cpacr_cp10 = CPACR.cp10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
                if NSACR.cp10 == '0' then cpacr_cp10 = '00';

            if PSTATE.EL != EL2 then
                // Check if Advanced SIMD disabled in CPACR
                if advsimd && cpacr_asedis == '1' then UNDEFINED;

                // Check if access disabled in CPACR
                case cpacr_cp10 of
                    when '00' disabled = TRUE;
                    when '01' disabled = PSTATE.EL == EL0;
                    when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                    when '11' disabled = FALSE;
                    if disabled then UNDEFINED;

                // If required, check FPEXC enabled bit.
                if fpexc_check && FPEXC.EN == '0' then UNDEFINED;
            AArch32.CheckFPAdvSIMDTrap(advsimd);  // Also check against HCPTR and CPTR_EL3
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if HaveEL(EL2) && !IsSecure() then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL(EL3) && !ELUsingAArch32(EL3) && !IsSecure() then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            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);

            // Check if access disabled in CPTR_EL3
            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() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            exception = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(exception);
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if HaveFGETExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                (HCR_EL2.<E2H,TGE> != '1' && (!HaveEL(EL1) || SCR_EL3.FGTEn == '1')));
        if route_to_el2 then
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

AArch32.CheckForWFXTrap(bits(2) target_el, boolean is_wfe)

// 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 !ELUsingAArch32(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() && !ELUsingAArch32(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 case;
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
if PSTATE.EL == EL2 then
  it_disabled = HSCTLR.ITD;
else
  it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
if it_disabled == '1' then
  if mask != '1000' then UNDEFINED;
  // Otherwise whether the IT block is allowed depends on hwl of the next instruction.
  next_instr = AArch32.MemSingle(NextInstrAddr(), 2, AccType_IFETCH, TRUE);
  if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx', '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
    // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
    // taken on the IT instruction or the next instruction. This is not reflected in
    // the pseudocode, which always takes the exception on the IT instruction. This
    // also does not take into account cases where the next instruction is UNPREDICTABLE.
    UNDEFINED;
return;

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
if AArch32.GeneralExceptionsToAArch64() then
  AArch64.CheckIllegalState();
elsif PSTATE.IL == '1' then
  route_to_hyp = PSTATE.EL == EL0 & EL2Enabled() & HCR.TGE == '1';
  bits(32) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x04;
  if PSTATE.EL == EL2 || route_to_hyp then
    exception = ExceptionSyndrome(Exception_IllegalState);
    if PSTATE.EL == EL2 then
      AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
      AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
  else
    AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETЕНEnabled

// AArch32.CheckSETЕНEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETЕНEnabled()
if PSTATE.EL == EL2 then
  setend_disabled = HSCTLR.SED;
else
  setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
if setend_disabled == '1' then
  UNDEFINED;
return;
// AArch32.SystemAccessTrap()
// =========================
// Trapped system register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL2
        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)
    exception;
    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTrap);
        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
        iss<13:10> = instr<19:16>;    // CRn, Reg in case of VMRS
        iss<8:5> = instr<15:12>;      // Rt
        iss<9> = '0';                  // RES0
        if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;    // opc2
            iss<16:14> = instr<23:21>;  // opc1
            iss<4:1> = instr<3:0>;      // CRm
        else    //VMRS Access
            iss<19:17> = '000';         //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';         //opc1 - Hardcoded for VMRS
            iss<4:1> = '0000';          //CRm - Hardcoded for VMRS
    elseif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;      // opc1
        iss<13:10> = instr<19:16>;    // Rt2
        iss<8:5> = instr<15:12>;      // Rt
        iss<4:1> = instr<3:0>;        // CRm
    elseif exception.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;      // imm8
        iss<4> = instr<23>;           // U
        iss<2:1> = instr<24,21>;      // P,W
        if instr<19:16> == '11111' then
            iss<8:5> = bits(4) UNKNOWN;
            iss<3> = '1';
        elseif exception.exceptype == Exception_Uncategorized then
            // Trapped for unknown reason
            iss<8:5> = instr<19:16>;     // Rn
            iss<3> = '0';
        else
            iss<0> = instr<20>;          // Direction
        end if
    end if
    if 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()
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);
elseif route_to_hyp then
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
    AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

// AArch32.UndefinedFault()
// ========================
AArch32.UndefinedFault()
  if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();
  AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/functions/aborts/AArch32.CreateFaultRecord

// AArch32.CreateFaultRecord()
// ===========================
FaultRecord AArch32.CreateFaultRecord(Fault statuscode, bits(40) ipaddress, bits(4) domain, integer level, AccType acctype, boolean write, bit extflag, bits(4) debugmoe, bits(2) errortype, boolean secondstage, boolean s2fs1walk)
  FaultRecord fault;
  fault.statuscode = statuscode;
  if (statuscode != Fault_None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fs1walk && AArch32.DomainValid(statuscode, level)) then
    fault.domain = domain;
  else
    fault.domain = bits(4) UNKNOWN;
  fault.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

// 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;
// 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 && (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.acctype == AccType_TTW &&
                boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk") ) ) then
            iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
    return iss;
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

bits(5) result;

case statuscode of
  when Fault_AccessFlag
    assert level IN {1,2};
    result = if level == 1 then '00011' else '00110';
  when Fault_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
  when Fault_ICacheMaint
    result = '00100';
  otherwise
    Unreachable();

return result;

// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

return imm32;
Library pseudocode for aarch32/functions/common/A32ExpandImm_C

// A32ExpandImm_C()
// ================
<bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UINT(imm12<11:8>), carry_in);
    return (imm32, carry_out);

Library pseudocode for aarch32/functions/common/DecodeImmShift

// DecodeImmShift()
// ================
<bits(2), 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()
// ================
<bits(2) srtype) DecodeRegShift<bits(2) srtype)

case srtype of
    when '00'  shift_t = SRTYPE_LSL;
    when '01'  shift_t = SRTYPE_LSR;
    when '10'  shift_t = SRTYPE_ASR;
    when '11'  shift_t = SRTYPE_ROR;

    return shift_t;

Library pseudocode for aarch32/functions/common/RRX

// RRX()
// =====
<bits(N) RRX<bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;

Library pseudocode for aarch32/functions/common/RRX_C

// RRX_C()
// ========
<bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<\N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
Library pseudocode for aarch32/functions/common/SRType

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};

Library pseudocode for aarch32/functions/common/Shift

// Shift()
// ========

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
  (result, -) = Shift_C(value, srtype, amount, carry_in);
  return result;

Library pseudocode for aarch32/functions/common/Shift_C

// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
  assert !(srtype == SRType_RRX && amount != 1);
  if amount == 0 then
    (result, carry_out) = (value, carry_in);
  else
    case srtype of
      when SRType_LSL
        (result, carry_out) = LSL_C(value, amount);
      when SRType_LSR
        (result, carry_out) = LSR_C(value, amount);
      when SRType_ASR
        (result, carry_out) = ASR_C(value, amount);
      when SRType_ROR
        (result, carry_out) = ROR_C(value, amount);
      when SRType_RRX
        (result, carry_out) = RRX_C(value, carry_in);
    return (result, carry_out);

Library pseudocode for aarch32/functions/common/T32ExpandImm

// 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

```plaintext
// T32ExpandImm_C()
// ================
(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

if imm12<11:10> == '00' then
    case imm12<9:8> of
    when '00' then
        imm32 = ZeroExtend(imm12<7:0>, 32);
    when '01' then
        imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
    when '10' then
        imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
    when '11' then
        imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
    end;
    carry_out = carry_in;
else
    unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
    (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));
end;
return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```plaintext
// 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, MCRR and MRRC disabled by HSTR<CRn/CRm>
    major = if nreg == 1 then CRn else CRm;
    if !(major IN {4,14}) && HSTR<major> == '1' then
        return TRUE;
    // 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)
{
    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = 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;
}

// 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);

// 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);
Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

// 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)
    accctype = AArch32.CheckAlignment(address, size, acctype, iswrite);
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;
    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
        MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
        AArch32.MarkExclusiveVA(address, ProcessorID(), size);

Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()
    fpexc_check = TRUE;
    advsimd = TRUE;
    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted
    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];
    return;

Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;

Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

// CheckCryptoEnabled32()
// ======================

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
Library pseudocode for aarch32/functions/float/CheckVFPEnabled

// CheckVFPEnabled()
// ===============

CheckVFPEnabled(boolean include_fpexc_check)
  advsimd = FALSE;
  AArch32_CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
  // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
  return;

Library pseudocode for aarch32/functions/float/FPHalvedSub

// FPHalvedSub()
// =============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
  assert N IN {16,32,64};
  rounding = FPRoundingMode(fpcr);
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
    if inf1 && inf2 && sign1 == sign2 then
      result = FPDefaultNaN();
    elsif inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
      result = FPIInfinity('0');
    elsif inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
      result = FPIInfinity('1');
    elsif zero1 && zero2 && sign1 != sign2 then
      result = FPZero(sign1);
    else
      result_value = (value1 - value2) / 2.0;
      if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr);
  return result;

Library pseudocode for aarch32/functions/float/FPRSqrtStep

// FPRSqrtStep()
// =============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
  assert N IN {16,32};
  FPCRType fpcr = StandardFPSCRValue();
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
    bits(N) product;
    if (inf1 && zero2) || (zero1 && inf2) then
      product = FPZero('0');
    else
      product = FPMul(op1, op2, fpcr);
    bits(N) three = FPTree('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);
  if (inf1 && zero2) || (zero1 && inf2) then
    product = FPZero('0');
  else
    product = FPMul(op1, op2, fpcr);
  bits(N) two = FPTwo('0');
  result = FPSub(two, product, fpcr);
return result;

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

// StandardFPSCRValue()
// ================

FPCRType StandardFPSCRValue()
  bits(32) upper = '00000000000000000000000000000000';
return upper : lower;

Library pseudocode for aarch32/functions/memory/AArch32.CheckAlignment

// AArch32.CheckAlignment()
// ================

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype, boolean iswrite)
if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
  A = SCTLR[].A; //use AArch64 register, when higher Exception level is using AArch64
elsif PSTATE.EL == EL2 then
  A = HSCLTR.A;
else
  A = SCTLR.A;
aligned = (address == Align(address, alignment));
atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                      AccType_ORDEREDATOMICRW, AccType_ATOMICLS64};
ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC,
                      AccType_ORDEREDATOMICRW, AccType_ATOMICLS64};
vector = acctype == AccType_VEC;
// 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] = bits(size*8) value;
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);
AddressDescriptor memaddrdesc;
bits(size*8) value;
iswrite = FALSE;

memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch32.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);
    value = _Mem[memaddrdesc, size, accdesc, FALSE];
return value;

// AArch32.MemSingle[] - assignment (write) form
// ----------------------------------------------------------------------------------
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) value;
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);
AddressDescriptor memaddrdesc;
iswrite = TRUE;

memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch32.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareable then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);
    _Mem[memaddrdesc, size, accdesc] = value;
return;

Library pseudocode for aarch32/functions/memory/Hint_PreloadData

Hint_PreloadData(bits(32) address);

Library pseudocode for aarch32/functions/memory/Hint_PreloadDataForWrite

Hint_PreloadDataForWrite(bits(32) address);
Library pseudocode for aarch32/functions/memory/Hint_PreloadInstr

Hint_PreloadInstr(bits(32) address);

Library pseudocode for aarch32/functions/memory/MemA

// MemA[] - non-assignment form
// ---------------------------------------------
bits(8*size) MemA[bits(32) address, integer size]
    acctype = AccType.Atomic;
    return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
// ---------------------------------------------
MemA[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType.Atomic;
    Mem_with_type[address, size, acctype] = value;
    return;

Library pseudocode for aarch32/functions/memory/MemO

// MemO[] - non-assignment form
// ---------------------------------------------
bits(8*size) MemO[bits(32) address, integer size]
    acctype = AccType.Ordered;
    return Mem_with_type[address, size, acctype];

// MemO[] - assignment form
// ---------------------------------------------
MemO[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType.Ordered;
    Mem_with_type[address, size, acctype] = value;
    return;

Library pseudocode for aarch32/functions/memory/MemU

// MemU[] - non-assignment form
// ---------------------------------------------
bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType.Normal;
    return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// ---------------------------------------------
MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType.Normal;
    Mem_with_type[address, size, acctype] = value;
    return;
Library pseudocode for aarch32/functions/memory/MemU_unpriv

// MemU_unpriv[] - non-assignment form
// ===================================
bits(8*size) MemU_unpriv[bits(32) address, integer size]
   acctype = AccType_UNPRIV;
   return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// ===============================
MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
   acctype = AccType_UNPRIV;
   Mem_with_type[address, size, acctype] = value;
   return;
// Mem_with_type[] - non-assignment (read) form
// ================================================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
assert size IN {1, 2, 4, 8, 16};
bits(size*8) value;
boolean iswrite = FALSE;

aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
if !aligned then
  assert size > 1;
  value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
  assert c IN {Constraint_FAULT, Constraint_NONE};
  if c == Constraint_NONE then aligned = TRUE;

  for i = 1 to size-1
    value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];

else

  value = AArch32.MemSingle[address, size, acctype, aligned];

  if BigEndian(acctype) then
    value = BigEndianReverse(value);

  return value;

// Mem_with_type[] - assignment (write) form
// ==============================================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
boolean iswrite = TRUE;

if BigEndian(acctype) then
  value = BigEndianReverse(value);

aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
if !aligned then
  assert size > 1;
  AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
  assert c IN {Constraint_FAULT, Constraint_NONE};
  if c == Constraint_NONE then aligned = TRUE;

  for i = 1 to size-1
    AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
else
  AArch32.MemSingle[address, size, acctype, aligned] = value;

return;
// AArch32.ESBOperation()
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && FL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.EA == '1';

if route_to_aarch64 then
    AArch64.ESBOperation();
    return;

route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.EA == '1';
route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE == '1' || HCR.AMO == '1');
if route_to_monitor then
target = M32_Monitor;
elsif route_to_hyp || PSTATE.M == M32_Hyp then
target = M32_Hyp;
else
target = M32_Abort;
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 that can be synchronized
// by an Error synchronization event.
if masked && IsSynchronizablePhysicalSErrorPending() 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();
Library pseudocode for aarch32/functions/ras/AArch32.ReportDeferredSError

```c
// 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

```c
type AArch32.SErrorSyndrome is (  
  bits(2) AET,  
  bit ExT  
);
```

Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```c
// 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() || ExternalDebugInterruptionsDisabled(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

// 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

// AArch32.ResetSIMDFPRegisters()
// ==============================

AArch32.ResetSIMDFPRegisters()
for i = 0 to 15
Q[i] = bits(128) UNKNOWN;
return;

Library pseudocode for aarch32/functionsregisters/AArch32.ResetSpecialRegisters

// AArch32.ResetSpecialRegisters()
// ===============================

AArch32.ResetSpecialRegisters()
// AArch32 special registers
SPSR_fiq<31:0> = bits(32) UNKNOWN;
SPSR_irq<31:0> = bits(32) UNKNOWN;
SPSR_svc<31:0> = bits(32) UNKNOWN;
SPSR_abt<31:0> = bits(32) UNKNOWN;
SPSR_und<31:0> = bits(32) UNKNOWN;
if HaveEL(EL2) then
SPSR_hyp = bits(32) UNKNOWN;
ELR_hyp = bits(32) UNKNOWN;
if HaveEL(EL3) then
SPSR_mon = bits(32) UNKNOWN;
// External debug special registers
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
return;

Library pseudocode for aarch32/functionsregisters/AArch32.ResetSystemRegisters

AArch32.ResetSystemRegisters(boolean cold_reset);
Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```plaintext
// ALUExceptionReturn()
// ================

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User, M32_System} then
        Constraint c = ConstrainUnpredictable(Unpredictable_ALU_EXCEPTIONRETURN);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF
                UNDEFINED;
            when Constraint_NOP
                EndOfInstruction();
        else
            AArch32.ExceptionReturn(address, SPSR[]);
```

Library pseudocode for aarch32/functions/registers/ALUWritePC

```plaintext
// 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

```plaintext
// 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' && ConstrainUnpredictableBool(Unpredictable_A32_FORCEALIGNPC) then
            address<1> = '0';
            BranchTo(address, branch_type);
```

Library pseudocode for aarch32/functions/registers/BranchWritePC

```plaintext
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address, BranchType branch_type)
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
        BranchTo(address, branch_type);
```
Library pseudocode for aarch32/functions/registers/D

// D[] - non-assignment form
// ========================================

bits(64) D[integer n]
   assert n >= 0 && n <= 31;
   base = (n MOD 2) * 64;
   bits(128) vreg = V[n DIV 2];
   return vreg<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
   assert n >= 0 && n <= 31;
   base = (n MOD 2) * 64;
   bits(128) vreg = V[n DIV 2];
   vreg<base+63:base> = value;
   V[n DIV 2] = vreg;
   return;

Library pseudocode for aarch32/functions/registers/Din

// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
   assert n >= 0 && n <= 31;
   return _Dclone[n];

Library pseudocode for aarch32/functions/registers/LR

// LR - assignment form
// ====================

LR = bits(32) value
   R[14] = value;
   return;

// LR - non-assignment form
// ========================

bits(32) LR
   return R[14];

Library pseudocode for aarch32/functions/registers/LoadWritePC

// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
   BXWritePC(address, BranchType_INDIR);
Library pseudocode for aarch32/functions/registers/LookUpRIndex

// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

case n of // Select index by mode:  usr  fiq  irq  svc  abt  und  hyp
    when 8     result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
    when 9     result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
    when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
    when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
    when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
    when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
    when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
    otherwise result = n;

return result;

Library pseudocode for aarch32/functions/registers/Monitor_mode_registers

bits(32) SP_mon;
bits(32) LR_mon;

Library pseudocode for aarch32/functions/registers/PC

// PC - non-assignment form
// ========================

bits(32) PC
return R[15];               // This includes the offset from AArch32 state

Library pseudocode for aarch32/functions/registers/PCStoreValue

// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    return PC;

Library pseudocode for aarch32/functions/registers/Q

// Q[] - non-assignment form
// =========================

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n] = value;
    return;
Library pseudocode for aarch32/functions/registers/Qin

// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];

Library pseudocode for aarch32/functions/registers/R

// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];

Library pseudocode for aarch32/functions/registers/RBankSelect

// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer 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 CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
    // register are unchanged or set to zero. This is also tested for on
    // exception entry, as this applies to all AArch32 registers.
    if !HighestELUsingAArch32() && 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> == '1111' then
        opc1 = instr<21+:3>;
        CRn = instr<16+:4>;
        CRm = instr<0+:4>;
        opc2 = instr<5+:3>;
        return (opc1 == '000' && CRn == '0111' && CRm == '1001' && opc2 IN {'000','001'});
    else
        return FALSE;
Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

// AArch32ExecutingCP10or11Instr()
// ================================

boolean AArch32ExecutingCP10or11Instr()

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

// AArch32ExecutingLSMInstr()
// ===========================
// Returns TRUE if processor is executing a Load/Store Multiple instruction

boolean AArch32ExecutingLSMInstr()

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()
// ===================

AArch32ITAdvance()

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 UsingAAArch32();
    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) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
  valid = FALSE;

if !valid then
  PSTATE.IL = '1';
else
  AArch32.WriteMode(mode);

// 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, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
  // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
  // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
  // AArch64, then these modes are EL1 modes.
  // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
  valid = HaveAArch32EL(EL1);
when M32_User
  valid = HaveAArch32EL(EL0);
otherwise
  valid = FALSE; // Passed an illegal mode value
return !valid;
Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

// BankedRegisterAccessValid()
// ========================================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

case SYSm of
    when '000xx', '00100'                          // R8_usr to R12_usr
        when mode != M32_FIQ then UNPREDICTABLE;
    when '00101'                                   // SP_usr
        when mode == M32_System then UNPREDICTABLE;
    when '00110'                                   // LR_usr
        when mode IN {M32_Hyp, M32_System} then UNPREDICTABLE;
    when '010xx', '0110x', '01110'                 // R8_fiq to R12_fiq, SP_fiq, LR_fiq
        when mode == M32_FIQ then UNPREDICTABLE;
    when '1000x'                                   // LR_irq, SP_irq
        when mode == M32_IRQ then UNPREDICTABLE;
    when '1001x'                                   // LR_svc, SP_svc
        when mode == M32_Svc then UNPREDICTABLE;
    when '1010x'                                   // LR_abt, SP_abt
        when mode == M32_Abort then UNPREDICTABLE;
    when '1011x'                                   // LR_und, SP_und
        when mode == M32_Undef then UNPREDICTABLE;
    when '1110x'                                   // LR_mon, SP_mon
        when !HaveEL(EL3) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;
    when '11110'                                   // ELR_hyp, only from Monitor or Hyp mode
        when !HaveEL(EL2) || !mode IN {M32_Monitor, M32_Hyp} then UNPREDICTABLE;
    when '11111'                                   // SP_hyp, only from Monitor mode
        when !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
    otherwise
        UNPREDICTABLE;

return;
Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

// CPSRWriteByInstr()
// ===============

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0; // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>; // Bits <26:24> are ignored
    end

    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        else if privileged then
            PSTATE.PAN = value<22>;
        end
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
            // Bit <20> is RES0
            PSTATE.GE = value<19:16>;
        end
    end

    if bytemask<1> == '1' then
        if HaveSSBSExt() then
            PSTATE.E = value<9>; // Bits <15:10> are RES0
        else if privileged then
            PSTATE.A = value<8>;
        end
    end

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);
        end
    end

    return;

Library pseudocode for aarch32/functions/system/ConditionPassed

// ConditionPassed()
// ===============

boolean ConditionPassed()
return ConditionHolds(AArch32.CurrentCond());

Library pseudocode for aarch32/functions/system/CurrentCond

bits(4) AArch32.CurrentCond();

Library pseudocode for aarch32/functions/system/InITBlock

// InITBlock()
// ===========

boolean InITBlock()
if CurrentInstrSet() == InstrSet_T32 then
    return PSTATE.IT<3:0> != '0000';
else
    return FALSE;

Library pseudocode for aarch32/functions/system/LastInITBlock

```plaintext
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```plaintext
// SPSRWriteByInstr()
// ================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    bits(32) new_spsr = SPSR[];
    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits
    if bytemask<2> == '1' then
    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask
    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits
    SPSR[] = new_spsr;                   // UNPREDICTABLE if User or System mode
    return;
```

Library pseudocode for aarch32/functions/system/SPSRaccessValid

```plaintext
// SPSRaccessValid()
// =============
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'                                                   // SPSR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '10000'                                                   // SPSR_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '10010'                                                   // SPSR_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '10100'                                                   // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110'                                                   // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'                                                   // SPSR_mon
            if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
        when '11110'                                                   // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;
    return;
```
Library pseudocode for aarch32/functions/system/SelectInstrSet

// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};
    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';
    return;

Library pseudocode for aarch32/functions/v6simd/Sat

// Sat()
// ======

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;

Library pseudocode for aarch32/functions/v6simd/SignedSat

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
// AArch32.CombineS1S2Desc()
// =========================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor AArch32.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc, AccType s2acctype)
{
    AddressDescriptor result;
    result.paddress = s2desc.paddress;

    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    else
        result.fault = AArch32.NoFault();
        if s2desc.memattrs.memtype == MemType_Device ||
           (!apply_force_writeback &&
            s1desc.memattrs.memtype == MemType_Device &&
            s2desc.memattrs.inner.attrs != '10') ||
           (!apply_force_writeback &&
            s1desc.memattrs.memtype == MemType_Device)
            then
            result.memattrs.memtype = MemType_Device;
        if s1desc.memattrs.memtype == MemType_Normal then
            result.memattrs.device = s2desc.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);
        result.memattrs.tagged = FALSE;
        // S1 can be either Normal or Device, S2 is Normal.
    else
        result.memattrs.memtype = MemType_Normal;
        result.memattrs.device = DeviceType_UNKNOWN;
        result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner,
                                                  s2desc.memattrs.inner,
                                                  s2desc.memattrs.device);
        result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer,
                                                  s2desc.memattrs.outer,
                                                  s2desc.memattrs.device);
        result.memattrs.shareable = (s1desc.memattrs.shareable ||
                                     s2desc.memattrs.shareable);
        result.memattrsoutershareable = (s1desc.memattrsoutershareable ||
                                           s2desc.memattrsoutershareable);
        result.memattrs.tagged = (s1desc.memattrs.tagged &&
                                     result.memattrsinner.attrs == MemAttr_WB &&
                                     result.memattrsinner.hints == MemHint_RWA &&
                                     result.memattrsouterattrs == MemAttr_WB &&
                                     result.memattrsouterhints == MemHint_RWA);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
}
// Library pseudocode for aarch32/translation/attrs/AArch32.DefaultTEXDecode

// AArch32.DefaultTEXDecode()
// ==========================

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011'
    then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB);
        TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;
    endcase

    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 '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(TEX<1:0>, acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints(TEX<1:0>, acctype, FALSE);
            memattrs.shareable = (S == '1');
        otherwise
            // Reserved, handled above
            Unreachable();
    endcase

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;

    // distinction between inner and outer shareable is not supported in this format
    memattrs.ctorshareable = memattrs.shareable;
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
Library pseudocode for aarch32/translation/attrs/AArch32.InstructionDevice

// 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 s2fslwalk)

    c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    assert c IN {Constraint_NONE, ConstraintFAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, level, acctype, iswrite,
                                                  secondstage, s2fslwalk);
    else
        addrdesc.memattrs.memtype = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs.tagged = FALSE;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;
// AArch32.RemappedTEXDecode()
// ===========================

MemoryAttributes AArch32.RemappedTEXDecode(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);
        end

        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();
        end

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
// AArch32.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

  MemoryAttributes memattrs;
  if PSTATE.EL == EL2 then
    mair = HMAIR1:HMAIR0;
  else
    mair = MAIR1:MAIR0;
  index = 8 * UInt(attr);
  attrfield = mair<index+7:index>;
  memattrs.tagged = FALSE;
  if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
      (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
    // Reserved, maps to an allocated value
    (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
  if !HaveMTEExt() && attrfield<7:4> == '1111' && attrfield<3:0> == '0000' then
    // Reserved, maps to an allocated value
    (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
  if attrfield<7:4> == '0000' then            // Device
    memattrs.memtype = MemType_Device;
    case attrfield<3:0> of
      when '0000' memattrs.device = DeviceType_nGnRnE;
      when '0100' memattrs.device = DeviceType_nGnRE;
      when '1000' memattrs.device = DeviceType_nGRE;
      when '1100' memattrs.device = DeviceType_GRE;
      otherwise Unreachable();         // Reserved, handled above
  elsif attrfield<3:0> != '0000'  then        // Normal
    memattrs.memtype = MemType_Normal;
    memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
    memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
    memattrs.shareable = SH<1> == '1';
    memattrsoutershareable = SH == '10';
  elsif HaveMTEExt() && attrfield == '11110000' then // Normal, Tagged WB-RWA
    memattrs.memtype = MemType_Normal;
    memattrs.outer = LongConvertAttrsHints('1111', acctype); // WB RWA
    memattrs.inner = LongConvertAttrsHints('1111', acctype); // WB_RWA
    memattrs.shareable = SH<1> == '1';
    memattrsoutershareable = SH == '10';
    memattrs.tagged = TRUE;
  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;
  result.descupdate.AF = FALSE;
  result.descupdate.AP = FALSE;

  default_cacheable = (HasS2Translation()) && ((if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC) == '1');
  if default_cacheable then
    // Use default cacheable settings
    result.addrdesc.memattrs.memtype = MemType_Normal;
    result.addrdesc.memattrs.inner.attrs = MemAttr_WB;  // Write-back
    result.addrdesc.memattrs.inner.hints = MemHint_RWA;
    result.addrdesc.memattrs.shareable = FALSE;
    result.addrdesc.memattrs.outershareable = FALSE;
    result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
  elseif acctype != AccType_IFETCH then
    // Treat data as Device
    result.addrdesc.memattrs.memtype = MemType_Device;
    result.addrdesc.memattrs.device = DeviceType_nGnRnE;
    result.addrdesc.memattrs.inner = MemAttrHints_UNKNOWN;
    result.addrdesc.memattrs.tagged = FALSE;
  else
    // Instruction cacheability controlled by SCTLR/HSCTLR.I
    if PSTATE.EL == EL2 then
      cacheable = HSCTLR.I == '1';
    else
      cacheable = SCTLR.I == '1';
    result.addrdesc.memattrs.memtype = MemType_Normal;
    if cacheable then
      result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
      result.addrdesc.memattrs.inner.hints = MemHint_RA;
    else
      result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
      result.addrdesc.memattrs.inner.hints = MemHint_No;
    result.addrdesc.memattrs.shareable = TRUE;
    result.addrdesc.memattrs.outershareable = TRUE;
    result.addrdesc.memattrs.tagged = FALSE;
  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();

  result.descupdate.descaddr = result.addrdesc;
  return result;
Library pseudocode for aarch32/translation/checks/AArch32.AccessUsesEL

```c
// AArch32.AccessUsesEL()
// ================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

Library pseudocode for aarch32/translation/checks/AArch32.CheckDomain

```c
// AArch32.CheckDomain()
// ===============

(boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level, AccType acctype, boolean iswrite)

    index = 2 * UInt(domain);
    attrfield = DACR<index+1:index>;

    if attrfield == '10' then          // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESDACR);

    if attrfield == '00' then
        fault = AArch32.DomainFault(domain, level, acctype, iswrite);
    else
        fault = AArch32.NoFault();

    permissioncheck = (attrfield == '01');

    return (permissioncheck, fault);
```
Library pseudocode for aarch32/translation/checks/AArch32.CheckPermission
FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level, bits(4) domain, bit NS, AccType accType, boolean iswrite)

assert ELUsingAArch32(S1TranslationRegime());

if PSTATE.EL != EL2 then
    wxn = SCTLR.WXN == '1';
    if TTBCR.EAE == '1' || SCTLR.AFE == '1' || perms.ap<0> == '1' then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';
    else
        priv_r = perms.ap<2:1> != '00';
        priv_w = perms.ap<2:1> == '01';
        user_r = perms.ap<1> == '1';
        user_w = FALSE;

    uwxn = SCTLR.UWXN == '1';
    ispriv = AArch32.AccessUsesEL(accType) != EL0;

    user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
    priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
        (priv_w && wxn) || (user_w && uwxn));
    pan = if HavePANExt() then PSTATE.PAN else '0';
    is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
    is_ats1xp = (acctype == AccType_AT && AArch32.ExecutingATS1xPInstr());
    if pan == '1' && user_r && ispriv && (is_ldst || is_ats1xp) then
        priv_r = FALSE;
        priv_w = FALSE;
    else
        (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2
        wxn = HSCTLR.WXN == '1';
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;

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 acctype == AccType_DC then
    // DC maintenance instructions operating by VA, cannot fault from stage 1 translation.
    fail = FALSE;
elsif iswrite then
    fail = !w;
    failedread = FALSE;
else
    fail = !r;
    failedread = TRUE;
if fail then
    secondstage = FALSE;
    s2fs1walk = FALSE;
### Library pseudocode for aarch32/translation/checks/AArch32.CheckS2Permission

// 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.xx == '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
Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
assert ELUsingAArch32(SITranslationRegime());
assert size IN {2,4};

match = FALSE;
mismatch = FALSE;

for i = 0 to UInt(DBGDIDR.BRPs)
  (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
  match = match || match_i;
  mismatch = mismatch || mismatch_i;

if match && HaltOnBreakpointOrWatchpoint() then
  reason = DebugHalt_Breakpoint;
  Halt(reason);
elsif (match || mismatch) then
  acctype = AccType_IFETCH;
  iswrite = FALSE;
  debugmoe = DebugException_Breakpoint;
  return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
  return AArch32.NoFault();

Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)
FaultRecord fault = AArch32.NoFault();
d_side = (acctype != AccType_IFETCH);
generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
halt = HaltOnBreakpointOrWatchpoint();
// Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);

if !d_side && vector_catch_first && generate_exception then
  fault = AArch32.CheckVectorCatch(vaddress, size);
if fault.statuscode == Fault_None && (generate_exception || halt) then
  if d_side then
    fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
  else
    fault = AArch32.CheckBreakpoint(vaddress, size);
if fault.statuscode == Fault_None && !d_side && !vector_catch_first && generate_exception then
  return AArch32.CheckVectorCatch(vaddress, size);

return fault;
Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```c
// AArch32.CheckVectorCatch()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(bits 32) vaddress, integer size
assert ELUsingAArch32(S1TranslationRegime());
match = AArch32.VCMatch(vaddress);
if size == 4 && !match && AArch32.VCMatch(vaddress + 2) then
  match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
if match then
  acctype = AccType_IFETCH;
iswrite = FALSE;
debugmoe = DebugException_VectorCatch;
return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
  return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```c
// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(bits 32) vaddress, AccType acctype,
  boolean iswrite, integer size
assert ELUsingAArch32(S1TranslationRegime());
if acctype IN {AccType_TTW, AccType_IC, AccType_AT} then
  return AArch32.NoFault();
if acctype == AccType_DC then
  if !iswrite then
    return AArch32.NoFault();
  elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
    return AArch32.NoFault();
match = FALSE;
ispriv = AArch32.AccessUsesEL(acctype) != EL0;
for i = 0 to UInt(DBGDDIDR.WRPs)
  match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite);
if match && HaltOnBreakpointOrWatchpoint() then
  reason = DebugHalt_Watchpoint;
  EDWAR = vaddress;
  Halt(reason);
elsif match then
  debugmoe = DebugException_Watchpoint;
  return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
  return AArch32.NoFault();
```
**Library pseudocode for aarch32/translation/faults/AArch32.AccessFlagFault**

```plaintext
// 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 ELUAsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE);
        dc = (if ELUAsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        s1_enabled = tge == '0' && dc == '0' && SCTLR.M == '1';
    else
        s1_enabled = SCTLR.M == '1';
    end

    TLBRecord S1;
    S1.addrdesc.fault = AArch32.NoFault();
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                         // First stage enabled
        use_long_descriptor_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
        if use_long_descriptor_format then
            S1 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage, s2fs1walk, size);
            permissioncheck = TRUE;  domaincheck = FALSE;
        else
            S1 = AArch32.TranslationTableWalkSD(vaddress, acctype, iswrite, size);
            permissioncheck = TRUE;  domaincheck = TRUE;
        end
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;  domaincheck = FALSE;
    end

    InGuardedPage = FALSE;                 // No memory is guarded when stage 1 address translation is disabled

    if !IsFault(S1.addrdesc) && UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() &&
    nTLSMD = if S1TranslationRegime() == EL2 then HSCTLR.nTLSMD else SCTLR.nTLSMD;
    if nTLSMD == '0' then
        S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
    end

    // Check for unaligned data accesses to Device memory
    if (((!wasaligned && acctype != AccType_IFETCH) || acctype == AccType_DCZVA) &&
    !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device && S1.addrdesc.memattrs.device != DeviceType_GENERIC) &&
    !IsFault(S1.addrdesc) && domaincheck && ((acctype IN (AccType_DC, AccType_DC_UNPRIV, AccType_IC))
    (permissioncheck, abort) = AArch32.CheckDomain(S1.domain, vaddress, S1.level, acctype, iswrite);

    S1.addrdesc.fault = abort;

    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, vaddress, S1.level,
        S1.domain, S1.addrdesc.paddress.NS, acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device &&
    acctype == AccType_IFETCH) then
        S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level, S1.domain, acctype, iswrite,
        secondstage, s2fs1walk);
    end

    return S1.addrdesc;
// AArch32.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch32.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && HasS2Translation() then
        s2fs1walk = FALSE;
        result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size);
    else
        result = S1;
    return result;
// 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);

    // Check for unaligned data accesses to Device memory
    if (((!wasaligned && acctype != AccType_IFETCH) ||
        (acctype == AccType_DCZVA && !s2fs1walk)) &&
        S2.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S2.addrdesc)) then
        S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

    // Check for permissions on Stage2 translations
    if !IsFault(S2.addrdesc) then
        S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level, acctype, iswrite, s2fs1walk);

    // Check for instruction fetches from Device memory not marked as execute-never. As there
    // has not been a Permission Fault then the memory is not marked execute-never.
    if (!s2fs1walk && !IsFault(S2.addrdesc) &&
        S2.addrdesc.memattrs.memtype == MemType_Device &&
        acctype == AccType_IFETCH) then
        domain = bits(4) UNKNOWN;
        S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level, domain, acctype, iswrite, secondstage, s2fs1walk);

if (s2fs1walk && !IsFault(S2.addrdesc) &&
    S2.addrdesc.memattrs.memtype == MemType_Device) then
    // Check for protected table walk.
    if HCR.PTW == '1' then
        domain = bits(4) UNKNOWN;
        S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype, iswrite, secondstage, s2fs1walk);
    else
        // Translation table walk occurs as Normal Non-cacheable memory.
        S2.addrdesc.memattrs.mtype = MemType_Normal;
        S2.addrdesc.memattrs.inner.attrs = MemAttr_NC;
        S2.addrdesc.memattrs.outer.attrs = MemAttr_NC;
        S2.addrdesc.memattrs.shareable = TRUE;
        S2.addrdesc.memattrs.outershareable = TRUE;

if s2fs1walk then
    result = AArch32.CombineS1S2Desc(S1, S2.addrdesc, AccType_TTW);
else
    result = AArch32.CombineS1S2Desc(S1, S2.addrdesc, acctype);
else
    result = S1;
return result;
// AArch32.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation table walk access.
AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    assert HasS2Translation();
    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk, size);

// AArch32.TranslateAddress()
// =========================
// Main entry point for translating an address
AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

    if !ELUsingAArch32(S1TranslationRegime()) then
        return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned, size);
    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);
    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);
    return result;
// AArch32.TranslationTableWalkLD()
// ================================
// Returns a result of a translation table walk using the Long-descriptor format
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress,
   AccType accotype, 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

result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;
domain = bits(4) UNKNOWN;
descaddr.memattrs.memtype = MemType_Normal;

// Fixed parameters for the translation table walk:
//  grainsize = Log2(Size of Table) - Size of Table is 4KB in AArch32
//  stride = Log2(Address per Level) - Bits of address consumed at each level
constant integer grainsize = 12; // Log2(4KB page size)
constant integer stride = grainsize - 3; // Log2(page size / 8 bytes)

// Derived parameters for the translation table walk:
//  inputsize = Log2(Size of Input Address) - Input Address size in bits
//  level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
   // First stage translation
   inputaddr = ZeroExtend(vaddress);
el = AArch32.AccessUsesEL(accotype);
isprivileged = AArch32.AccessUsesEL(accotype) != EL0;
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, HTCRORGN0, HTCRIRGN0, secondstage);
   reversedescriptors = HSCTRL.EE == '1';
   lookupsecure = FALSE;
   singlepriv = TRUE;
   hierattrssdisabled = AArch32.HaveHPDExt() & & HTCROPD == '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.S0, TTBCR.ORGN0, TTBCR.IRGN0, secondstage);
   hierattrssdisabled = AArch32.HaveHPDExt() & & TTBCR.T2E == '1' & & TTBCR2.PD == '1';
tlsize = UInt(TTBCR.T1SZ);
if (tlsize == 0 & & !basefound) || (tlsize > 0 & & IsOnes(inputaddr<31:(32-tlsize)>)) then
   inputsize = 32 - tlsize;
   basefound = TRUE;
   baseregister = TTBR1;
descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGN1, TTBCR.IRGN1, secondstage);
hierattrsd-disabled = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD1 == '1';
reversedescriptors = SCTLR.EE == '1';
lookupsecure = IsSecure();
singlepriv = FALSE;
  // The starting level is the number of strides needed to consume the input address
  level = 4 - (1 + ((inputsize - grainsize - 1) DIV stride));
else
  // Second stage translation
  inputaddr = ipaddress;
  inputsize = 32 - SInt(VTCR.T0SZ);
  // VTCR.S must match VTCR.T0SZ[3]
  if VTCR.S != VTCR.T0SZ<3> then
    (-, inputsize) = ConstrainUnpredictableInteger(32-7, 32+8, Unpredictable_RESVTCRS);
  basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
  disabled = FALSE;
  descaddr.memattrs = WalkAttrDecode(VTCR.SH0, VTCR.ORGN0, VTCR.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 !basefound || disabled then
    level = 2;           // 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(Num of entries*8)
  // baseaddress = baseregister<39:baselowerbound>:Zeros(baselowerbound);

  ns_table = if lookupsecure then '0' else '1';
  ap_table = '00';
  xn_table = '0';
  pxn_table = '0';

  addrselecttop = inputsize - 1;
  repeat
    addrselectbottom = (3-level)*stride + grainsize;
    bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.address = ZeroExtend(baseaddress OR index);
    descaddr.paddress.NS = ns_table;
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
    if level <= 0 then basefound = FALSE;
  until level > 0;
/ If there are two stages of translation, then the first stage table walk addresses
// are themselves subject to translation
if secondstage || HasS2Translation() then
descaddr2 = descaddr;
else
descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
// Check for a fault on the stage 2 walk
if IsFault(descaddr2) then
    result.addrdesc.fault = descaddr2.fault;
return result;

// Update virtual address for abort functions
descaddr2.vaddress = ZeroExtend(vaddress);
accdesc = CreateAccessDescriptorTTW(acctype, secondstage, s2fs1walk, level);
desc = _Mem[descaddr2, 8, accdesc, iswrite];
if reversedescriptors then desc = BigEndianReverse(desc);
if desc[0] == '0' || (desc[1:0] == '01' && level == 3) then
    // Fault (00), Reserved (10), or Block (01) at level 3.
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                        iswrite, secondstage, s2fs1walk);
    return result;

// Valid Block, Page, or Table entry
if desc[1:0] == '01' || level == 3 then
    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

            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]; // Bit[54] of the block/page descriptor holds UXN
    pxn = desc[53]; // Bit[53] of the block/page descriptor holds PXN
    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
    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if !singlepriv 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
        result.GP = desc<50>;
        result.perms.ap<0>   = '1';
        result.addrdesc.memattrs = AArch32.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 s2fs1walk then
            result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_TTW);
        else
            result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
            result.addrdesc.paddress.NS = '1';
        end
        result.addrdesc.paddress.address = ZeroExtend(outputaddress);
        result.addrdesc.fault = AArch32.NoFault();
        result.contiguous = contiguousbit == '1';
        if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;
    end
end

return result;
// AArch32.TranslationTableWalkSD()
// =================================
// Returns a result of a translation table walk using the Short-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)
assert ELUsingAArch32(S1TranslationRegime());

// This is only called when address translation is enabled
TLBRecord result;
AddressDescriptor l1descaddr;
AddressDescriptor l2descaddr;
bits(40) outputaddress;

// Variables for Abort functions
ipaddress = bits(40) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;
NS = bit UNKNOWN;

// Default setting of the domain and level.
domain = bits(4) UNKNOWN;
level = 1;

// Determine correct Translation Table Base Register to use.
bits(64) ttbr;
n = UInt(TTBCR.N);
if n == 0 || IsZero(vaddress<31:(32-n)>)
  ttbr = TTBR0;
  disabled = (TTBCR.PD0 == '1');
else
  ttbr = TTBR1;
  disabled = (TTBCR.PD1 == '1');
  n = 0; // TTBR1 translation always works like N=0 TTBR0 translation

// Check if Translation table walk disabled for translations with this Base register.
if disabled then
  result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
  return result;

// Obtain descriptor from initial lookup.
l1descaddr.paddress.address = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
IRGN = ttbr<0>:ttbr<6>; // TTBR.IRGN
RGN = ttbr<4:3>; // TTBR.RGN
SH = ttbr<1>:ttbr<5>; // TTBR.S:TTBR.NOS
l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN, secondstage);
if !HasS2Translation() then
  // if only 1 stage of translation
  l1descaddr2 = l1descaddr;
else
  l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
  // Check for a fault on the stage 2 walk
  if IsFault(l1descaddr2) then
    result.addrdesc.fault = l1descaddr2.fault;
    return result;

// Update virtual address for abort functions
l1descaddr2.vaddress = ZeroExtend(vaddress);

accdesc = CreateAccessDescriptorTTW(acctype, secondstage, s2fs1walk, level);
l1desc = _Mem[l1descaddr2, 4, accdesc, iswrite];
if SCTLR.EE == '1' then l1desc = BigEndianReverse(l1desc);
// Process descriptor from initial lookup.
case l1desc<1:0> of
    when '00' // Fault, Reserved
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
        return result;
    when '01' // Large page or Small page
        domain = l1desc<8:5>;
        level = 2;
        pxn = l1desc<2>;
        NS = l1desc<3>;

        // Obtain descriptor from level 2 lookup.
        l2descaddr.paddress.address = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
        l2descaddr.paddress.NS = if isSecure() then '0' else '1';
        l2descaddr.memattrs = l1descaddr.memattrs;
        if !HaveEL(EL2) || (isSecure() && !isSecureEL2Enabled()) then
            // if only 1 stage of translation
            l2descaddr2 = l2descaddr;
        else
            l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);
            // Check for a fault on the stage 2 walk
            if IsFault(l2descaddr2) then
                result.addrdesc.fault = l2descaddr2.fault;
                return result;
        end

        // Update virtual address for abort functions
        l2descaddr2.vaddress = ZeroExtend(vaddress);
        accdesc = CreateAccessDescriptorTTW(acctype, secondstage, s2fs1walk, level);
        l2desc = _Mem[l2descaddr2, 4, accdesc, iswrite];
        if SCTLR.EE == '1' then l2desc = BigEndianReverse(l2desc);

        // Process descriptor from level 2 lookup.
        if l2desc<1:0> == '00' then
            result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
            return result;
        end

        nG = l2desc<11>;
        S = l2desc<10>;
        ap = l2desc<9,5:4>;

        if SCTLR.AFE == '1' && l2desc<4> == '0' then
            // Armv8 VMSAv8-32 does not support hardware management of the Access flag.
            result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
            return result;
        end

        if l2desc<1> == '0' then // Large page
            xn = l2desc<15>;
            tex = l2desc<14:12>;
            c = l2desc<3>;
            b = l2desc<2>;
            blocksize = 64;
            outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);
        else // Small page
            tex = l2desc<8:6>;
            c = l2desc<3>;
            b = l2desc<2>;
            xn = l2desc<0>;
            blocksize = 4;
            outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);
        end

    when '1x' // Section or Supersection
        NS = l1desc<19>;

    when '11' // Fault, Reserved
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
        return result;
end
nG = l1desc<17>;  
S = l1desc<16>;  
ap = l1desc<15,11:10>;  
tex = l1desc<14:12>;  
 txn = l1desc<4>;  
c = l1desc<3>;  
b = l1desc<2>;  
pxn = l1desc<0>;  
level = 1;
if SCTLR.AFE == '1' && l1desc<10> == '0' then  
   // Armv8 VMSAv8-32 does not support hardware management of the Access flag.
   result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, 
iswrite, secondstage, s2fs1walk);
   return result;
if 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.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 !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, DBGCR_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 aarch64/debug/breakpoint/AArch64.BreakpointValueMatch
boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    // matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    if n > UInt(ID_AA64DFR0_EL1.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs), Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking).
    if DBGBCR_EL1[n].E == '0' then return FALSE;
    context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));
    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR_EL1[n].BT;

    if ((dbgtype IN {'011x','11xx'}) && !HaveVirtHostExt() && !HaveV82Debug()) ||    // 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 {ConstraintDISABLED, Constraint_UNKNOWN};
        if c == ConstraintDISABLED 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
            byte = 0;                            // "vaddress" is word aligned
            byte_select_match = TRUE;            // DBGBCR_EL1[n].BAS-byte> is RES1
            // If the DBGxVR<n> EL1.RESS field bits are not a sign extension of the MSB
            // of DBGVR<n> EL1.VA, it is UNPREDICTABLE whether they appear to be
            // included in the match.
            // If 'vaddress' is outside of the current virtual address space, then the access
            // generates a Translation fault.
            integer top = if Have52BitVAExt() then 52 else 48;
            if !IsOnes(DBGBVr EL1[n]<63:top>) && !IsZero(DBGBVr EL1[n]<63:top>) then
                if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                    top = 63;
        else
            byte = 0;
        // "vaddress" is word aligned
        byte_select_match = TRUE;            // DBGBCR_EL1[n].BAS-byte> is RES1
        // If the DBGxVR<n> EL1.RESS field bits are not a sign extension of the MSB
        // of DBGVR<n> EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        // If 'vaddress' is outside of the current virtual address space, then the access
        // generates a Translation fault.
        integer top = if Have52BitVAExt() then 52 else 48;
        if !IsOnes(DBGBVr EL1[n]<63:top>) && !IsZero(DBGBVr EL1[n]<63:top>) then
            if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                top = 63;
BVR_match = (vaddress<top:2> == DBGVR_EL1[n]<top:2>) && byte_select_match;

elsif match_cid then
    if IsInHost() then
        BVR_match = (CONTEXTIDR_EL2<31:0> == DBGVR_EL1[n]<31:0>);
    else
        BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGVR_EL1[n]<31:0>);
    end if
elsif match_cid1 then
    BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1<31:0> == DBGVR_EL1[n]<31:0>);
endif

if match_vmid then
    if !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGVR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGVR_EL1[n]<47:32>;
    end if
    BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() && vmid == bvr_vmid);
elsif match_cid2 then
    BXVR_match = ((HaveVirtHostExt() || HaveV82Debug()) && EL2Enabled() &&
                    DBGVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);
endif

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);
mach = (!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 CONSTRANDED 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
            when Constraint_DISABLED return FALSE;                     // Disabled
            when Constraint_NONE linked = FALSE;                       // No linking
        end if
        // 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
        return priv_match && security_state_match && (!linked || linked_match);
Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

// AArch64.GenerateDebugExceptions()
// ---------------------------------

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

// AArch64.GenerateDebugExceptionsFrom()
// -------------------------------------

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)
    if OSLR_EL1.OSLK == '1' || DoubleLockStatus() || Halted()
        return FALSE;
    route_to_el2 = HaveEL(EL2) && (!secure || IsSecureEL2Enabled()) && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
target = (if route_to_el2 then EL2 else EL1);
    enabled = !HaveEL(EL3) || !secure || MDCR_EL3.SDD == '0';
    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);
    return enabled;

Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

// AArch64.CheckForPMUOverflow()
// ------------------------------

// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()
    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1';
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL(EL2) 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' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;
    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSLR_EL0.)
    return pmuirq;
Library pseudocode for aarch64/debug/pmu/AArch64.CountEvents
boolean AArch64.CountEvents(integer n)
assert n == 31 || n <.UInt(PMCR_EL0.N);

// Event counting is disabled in Debug state
debug = Halted();

// In Non-secure state, some counters are reserved for EL2
if HaveEL(EL2) then
    resvd_for_el2 = n >= UInt(MDCR_EL2.HPMN) && n != 31;
else
    resvd_for_el2 = FALSE;

// Main enable controls
E = if resvd_for_el2 then MDCR_EL2.HPME else PMCR_EL0.E;
enabled = E == '1' && PMCNTENSET_EL0<n> == '1';

// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;

// Event counting in Secure state is prohibited if all of:
// * EL3 is implemented
// * MDCR_EL3.SPME == 0, and either:
//   - FEAT_PMUv3p7 is not implemented
//   - MDCR_EL3.MPMX == 0
if HaveEL(EL3) && IsSecure() then
    if HavePMUv3p7() then
        prohibited = MDCR_EL3.<SPME,MPMX> == '00';
    else
        prohibited = MDCR_EL3.SPME == '0';

// Event counting at EL3 is prohibited if all of:
// * FEAT_PMUv3p7 is implemented
// * One of the following is true:
//   - MDCR_EL3.SPME == 1
//   - PMNx is not reserved for EL2
// * MDCR_EL3.MPMX == 1
if !prohibited && PSTATE.EL == EL3 && HavePMUv3p7() then
    prohibited = MDCR_EL3.<SPME,MPMX> == '1' || !resvd_for_el2);

// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * MDCR_EL2.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
    prohibited = MDCR_EL2.HPMD == '1';

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// PMCR_EL0.DP disables the cycle counter when event counting is prohibited
if enabled && prohibited && n == 31 then
    enabled = PMCR_EL0.DP == '0';

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if Havev85PMU() && n == 31 then
    if HaveEL(EL3) && IsSecure() && MDCR_EL3.SCCD == '1' then
        prohibited = TRUE;
    if PSTATE.EL == EL2 && MDCR_EL2.HCCD == '1' then
        prohibited = TRUE;

// If FEAT_PMUv3p7 is implemented, cycle counting an be prohibited at EL3.
// This is not overridden by PMCR_EL0.DP.
if HavePMUv3p7() && n == 31 then
    if PSTATE.EL == EL3 && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;
// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() && n != 31 then
    ovflw = PMOVSCLR_EL0<UInt(PMCR_EL0.N)-1:0>;
    if resvd_for_el2 then
        FZ = MDCR_EL2.HPMFZO;
        ovflw<UInt(MDCR_EL2.HPMN)-1:0> = Zeros();
    else
        FZ = PMCR_EL0.FZO;
        if HaveEL(EL2) then
            ovflw<UInt(PMCR_EL0.N)-1:UInt(MDCR_EL2.HPMN)> = Zeros();
        else
            ovflw<UInt(MDCR_EL2.HPMN)-1:0> = Zeros();
    end if;

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} bits
    filter = if n == 31 then PMCCFILTR_EL0[31:0] else PMEVTYPER_EL0[n]<31:0>;
    P   = filter<31>;
    U   = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M   = if HaveEL(EL3) then filter<26> else '0';
    SH  = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';

    case PSTATE.EL of
        when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
        when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
        when EL2 filtered = if IsSecure() then NSH == SH else NSH == '0';
        when EL3 filtered = M != P;
    end case;

    return !debug && enabled && !prohibited && !filtered && !frozen;

Library pseudocode for aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess

// CheckProfilingBufferAccess()
// --------------------------------------------
SysRegAccess CheckProfilingBufferAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;
    if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.E2PB<0> != '1' then
        return SysRegAccess_TrapToEL2;
    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;
    return SysRegAccess_OK;

Library pseudocode for aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess

// CheckStatisticalProfilingAccess()
// ----------------------------------
SysRegAccess CheckStatisticalProfilingAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;
    if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.TPMS == '1' then
        return SysRegAccess_TrapToEL2;
    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;
    return SysRegAccess_OK;
Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

// CollectContextIDR1()
// -------------------

boolean CollectContextIDR1()
if !StatisticalProfilingEnabled() then return FALSE;
if PSTATE.EL == EL2 then return FALSE;
if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
return 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)) || IsSecureEL2Enabled()) then
    return PMSCR_EL2.PA == '1' && (el == EL2 || PMSCR_EL1.PA == '1');
else
    return PMSCR_EL1.PA == '1';

Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

// CollectTimeStamp()
// ------------------

TimeStamp CollectTimeStamp()
if !StatisticalProfilingEnabled() then return TimeStamp_None;
(secure, el) = ProfilingBufferOwner();
if el == EL2 then
    if PMSCR_EL2.TS == '0' then return TimeStamp_None;
else
    if PMSCR_EL1.TS == '0' then return TimeStamp_None;
if EL2Enabled() then
    case PMSCR_EL2.PCT of
    when '00' return TimeStamp_Virtual;
    when '01' if el == EL2 then return TimeStamp_Virtual;
    when '11' if (el == EL2 || PMSCR_EL1.PCT != '00') && HaveECVExt() then
        return TimeStamp_OffsetPhysical;
    otherwise
        Unreachable();
    end case;
    end if;
    case PMSCR_EL1.PCT of
    when '00' return TimeStamp_Virtual;
    when '01' return TimeStamp_Physical;
    when '11' if HaveECVExt() then return TimeStamp_OffsetPhysical;
    otherwise
        Unreachable();
end case;
**Library pseudocode for aarch64/debug/statisticalprofiling/OpType**

```plaintext
enumeration OpType {
  OpType_Load,       // Any memory-read operation other than atomics, compare-and-swap, and swap
  OpType_Store,      // Any memory-write operation, including atomics without return
  OpType_LoadAtomic, // Atomics with return, compare-and-swap and swap
  OpType_Branch,     // Software write to the PC
  OpType_Other       // Any other class of operation
};
```

**Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled**

```plaintext
// 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' && PMSR_EL1.S == '0');
}
```

**Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner**

```plaintext
// 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**

```plaintext
// 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();
```
// SPECollectRecord()
// ==================
// Returns TRUE if the sampled class of instructions or operations, as
determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECollectRecord(bits(64) events, integer total_latency, OpType optype)
assert StatisticalProfilingEnabled();

bits(64) mask = 0xAA<63:0>; // Bits [7,5,3,1]
if HaveSVE() then mask<18:17> = Ones(); // Predicate flags
if HaveStatisticalProfilingv1p1() then mask<11> = '1'; // Alignment Flag
if HaveStatisticalProfilingv1p2() then mask<6> = '1'; // Not taken flag
mask<63:48> = bits(16) IMPLEMENTATION_DEFINED;
mask<31:24> = bits(8) IMPLEMENTATION_DEFINED;
mask<15:12> = bits(4) IMPLEMENTATION_DEFINED;

  // Check for UNPREDICTABLE case
if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.<FnE,FE> == '11' &&
!IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
    if ConstrainUnpredictableBool(Unpredictable_BADPMSFCR) then
      return FALSE;
else
  // Filtering by event
if PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
  e = events AND mask;
  m = PMSEVFR_EL1 AND mask;
  if !IsZero(NOT(e) AND m) then return FALSE;
  // Filtering by inverse event
if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.FnE == '1' &&
!IsZero(PMSNEVFR_EL1)) then
  e = events AND mask;
  m = PMSNEVFR_EL1 AND mask;
  if !IsZero(e AND m) then return FALSE;

  // Filtering by type
if PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>) then
  case optype of
  when OpType_Branch if PMSFCR_EL1.B == '0' then return FALSE;
  when OpType_Load if PMSFCR_EL1.LD == '0' then return FALSE;
  when OpType_STORE if PMSFCR_EL1.ST == '0' then return FALSE;
  when OpType_LoadAtomic if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
  otherwise
      return FALSE;
  // Filtering by latency
if PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT) then
  if total_latency < UInt(PMSLATFR_EL1.MINLAT) then
    return FALSE;

  // Check for UNPREDICTABLE cases
if ((PMSFCR_EL1.FE == '1' && IsZero(PMSEVFR_EL1 AND mask)) ||
(PMSFCR_EL1.FT == '1' && IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
(PMSFCR_EL1.FL == '1' && IsZero(PMSLATFR_EL1.MINLAT))) then
  return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

if (HaveStatisticalProfilingv1p2() &&
  ((PMSFCR_EL1.FE == '1' && IsZero(PMSNEVFR_EL1 AND mask)) ||
  (PMSFCR_EL1.<FnE,FE> == '11' &&
  !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)))) then
  return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

return TRUE;
Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```java
// StatisticalProfilingEnabled()
// =============================
boolean StatisticalProfilingEnabled()
  if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
    return FALSE;
  in_host = EL2Enabled() && HCR_EL2.TGE == '1';
  (secure, el) = ProfilingBufferOwner();
  if UInt(el) < UInt(PSTATE.EL) || secure != IsSecure() || (in_host && el == EL1) then
    return FALSE;
  case PSTATE.EL of
    when EL3 Unreachable();
    when EL2 spe_bit = PMSCR_EL2.E2SPE;
    when EL1 spe_bit = PMSCR_EL1.E1SPE;
    when EL0 spe_bit = (if in_host then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);
  return spe_bit == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/SysRegAccess

```java
enumeration SysRegAccess {
  SysRegAccess_OK,
  SysRegAccess_UNDEFINED,
  SysRegAccess_TrapToEL1,
  SysRegAccess_TrapToEL2,
  SysRegAccess_TrapToEL3
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```java
enumeration TimeStamp {
  TimeStamp_None, // No timestamp
  TimeStamp_CoreSight, // CoreSight time (IMPLEMENTATION DEFINED)
  TimeStamp_Physical, // Physical counter value with no offset
  TimeStamp_OffsetPhysical, // Physical counter value minus CNTPOFF_EL2
  TimeStamp_Virtual }; // Physical counter value minus CNTVOFF_EL2
```
AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && Uint(target_el) >= Uint(PSTATE.EL);
    sync_errors = HaveIESB() && SCTLR[target_el].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && target_el == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(UnpredictableIESBinDebug) then
        sync_errors = FALSE;
    SynchronizeContext();
    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    if from_32 then
        AArch64.MaybeZeroRegisterUppers();
        MaybeZeroSVEUppers(target_el);
    AArch64.ReportException(exception, target_el);
    PSTATE.L = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    SPSR[] = bits(64) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;
    if PSTATE.<SS,D,A,I,F> are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0'; // PSTATE.J is RES0
        if HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
            SCTLR[].SPAN == '0') then
            PSTATE.PAN = '1';
        if HaveUAOExt() then PSTATE.UAO = '0';
        if HaveBTIExt() then PSTATE.BTYPE = '00';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = '1';
        DLR_EL0 = bits(64) UNKNOWN;
        DSPSR_EL0 = bits(64) UNKNOWN;
        EDSCR.ERR = '1';
        UpdateEDSCRFields(); // Update EDSCR processor state flags.
        if sync_errors then
            SynchronizeErrors();
        EndOfInstruction();
Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n, AccType accType, bits(64) vaddress)

    integer top = if Have52BitVAExt() then 52 else 48;
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3; // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', 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_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

            if mask > bottom then
                // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
                // of DBGBVr<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
                // included in the match.
                if !IsOnes(DBGBVr_EL1[n]<63:top>) && !IsZero(DBGBVr_EL1[n]<63:top>) then
                    if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                        top = 63;
                        WVR_match = (vaddress<top:mask> == DBGWVR_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>;
                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($1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.WRPn);

    // "ispriv" is:
    //  * FALSE for all loads, stores, and atomic operations executed at EL0.
    //  * FALSE if the access is unprivileged.
    //  * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC, linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
    ls_match = FALSE;
    if acctype == AccType_ATOMICRW then
        ls_match = (DBGWCR_EL1[n].LSC != '0');
    else
        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);
            endif
        else
            AArch64.WatchpointException(vaddress, fault);
        endif
    elseif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```c
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(64) vaddress) {
    exception = ExceptionSyndrome(exctype);
    d_side = exceptype IN (Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_NV2Watchpoint);
    (exception.syndrome, exception.syndrome2) = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;
    return exception;
}
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```c
// AArch64.CheckPCAlignment()
// =========================
AArch64.CheckPCAlignment() {
    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
}
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```c
// 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' && HCR_EL2.TGA == '1' || HaveRASExt() && 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);
}
```
// AArch64.EffectiveTCF()
// ======================
// Returns the TCF field applied to tag check faults in the given Exception Level.

bits(2) AArch64.EffectiveTCF(bits(2) el)

  bits(2) tcf;

  if el == EL3 then
    tcf = SCTLR_EL3.TCF;
  elsif el == EL2 then
    tcf = SCTLR_EL2.TCF;
  elsif el == EL1 then
    tcf = SCTLR_EL1.TCF;
  elsif el == EL0 && HCR_EL2.<E2H,TGE> == '11' then
    tcf = SCTLR_EL2.TCF0;
  elsif el == EL0 && HCR_EL2.<E2H,TGE> != '11' then
    tcf = SCTLR_EL1.TCF0;

  return tcf;

// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)

  // External aborts on instruction fetch must be taken synchronously
  if HaveDoubleFaultExt() then assert fault.statuscode != Fault_AsyncExternal;
  route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
  route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                  (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                   (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)))));

  bits(64) preferred_exception_return = ThisInstrAddr();

  if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
      IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
    vect_offset = 0x180;
  else
    vect_offset = 0x0;

  exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

  if PSTATE.EL == EL3 || route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
  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.RaiseTagCheckFault

// AArch64.RaiseTagCheckFault()
// ============================
// Raise a tag check fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, boolean write)

bits(2) target_el;
bits(64) preferred_exception_return = ThisInstrAddr();
integer vect_offset = 0x0;

if PSTATE.EL == EL0 then
   target_el = if HCR_EL2.TGE == '0' then EL1 else EL2;
else
   target_el = PSTATE.EL;

exception = ExceptionSyndrome(Exception_DataAbort);
exception.syndrome<5:0> = '010001';
if write then
   exception.syndrome<6> = '1';
exception.vaddress = bits(4) UNKNOWN : va<59:0>;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

// AArch64.ReportTagCheckFault()
// =============================
// Records a tag check fault exception into the appropriate TCFR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    if el == EL3 then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
    elsif el == EL2 then
        if ttbr == '0' then
            TFSR_EL2.TF0 = '1';
        else
            TFSR_EL2.TF1 = '1';
        end
    elsif el == EL1 then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
        end
    elsif el == EL0 then
        if ttbr == '0' then
            TFSR_EL0.TF0 = '1';
        else
            TFSR_EL0.TF1 = '1';
        end

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);
// AArch64.TagCheckFault()
// =======================
// Handle a tag check fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccType acctype, boolean iswrite)
    bits(2) tcf = AArch64.EffectiveTCF(PSTATE.EL);
    case tcf of
        when '00'       // Tag Check Faults have no effect on the PE
            return;
        when '01'       // Tag Check Faults cause a synchronous exception
            AArch64.RaiseTagCheckFault(vaddress, iswrite);
        when '10'       // Tag Check Faults are asynchronously accumulated
            AArch64.ReportTagCheckFault(PSTATE.EL, vaddress<55>);
        when '11'       // Tag Check Faults cause a synchronous exception on reads or on
            // a read-write access, and are asynchronously accumulated on writes
            // Check for access performing both a read and a write.
            readwrite = acctype IN {AccType_ATOMICRW,
                AccType_ORDEREDATOMICRW,
                AccType_ORDEREDRW};
            if !iswrite || readwrite then
                AArch64.RaiseTagCheckFault(vaddress, iswrite);
            else
                AArch64.ReportTagCheckFault(PSTATE.EL, vaddress<55>);
    end;

// 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.FMO == '1');
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x100;
exception = ExceptionSyndrome(Exception_FIQ);

if route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
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()
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()
route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
            (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1');
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;
exception = ExceptionSyndrome(Exception_IRQ);

if route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
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() &&
( HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x180;
if IsSErrorEdgeTriggered(syndrome) then
  ClearPendingPhysicalSError();
exception = ExceptionSyndrome(Exception_SError);
exception.syndrome<24> = if impdef_syndrome then '1' else '0';
exception.syndrome<23:0> = syndrome;
if PSTATE.EL == EL3 || route_to_el3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elseif PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// -------------------------------

AArch64.TakeVirtualFIQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x100;
exception = ExceptionSyndrome(Exception_FIQ);
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// -------------------------------

AArch64.TakeVirtualIRQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;
exception = ExceptionSyndrome(Exception_IRQ);
AAArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```
Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

```c
// 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(ExceptionSError);
if HaveRASExt() then
    exception.syndrome<24> = VESR_EL2.IDS;
    exception.syndrome<23:0> = VESR_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

```c
// 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

```c
// 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';
    exception.syndrome<5:0> = '100010'; // IFSC = Debug Exception

if PSTATE.EL == EL2 || route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/debug/AArch64.VectorCatchException

// AArch64.VectorCatchException()
// -----------------------------------------------
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
assert PSTATE.EL != EL3;
assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

bits(64) preferred_exception_return = ThisInstrAddr();
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

```c
// 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_is_valid = TRUE;
    from_32 = UsingAArch32();

    case exceptype of
        when Exception_Uncategorized    ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap          ec = 0x01;    il_is_valid = FALSE;
        when Exception_CP15RTTrap        ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap       ec = 0x04; assert from_32;
        when Exception_CP14RTTrap        ec = 0x06; assert from_32;
        when Exception_CP14DRTTrap       ec = 0x07;    il_is_valid = FALSE;
        when Exception_AdvSIMDFPAccessTrap ec = 0x08;
        when Exception_FPIDTrap         ec = 0x09;
        when Exception_PACTrap          ec = 0x0A;
        when Exception_LPST64BTrap       ec = 0x0C; assert from_32;
        when Exception_IllegalState      ec = 0x0D;    il_is_valid = FALSE;
        when Exception_SupervisorCall    ec = 0x11;
        when Exception_HypervisorCall    ec = 0x12;
        when Exception_MonitorCall      ec = 0x13;
        when Exception_SystemRegisterTrap ec = 0x18; assert !from_32;
        when Exception_ERetTrap          ec = 0x19; assert !from_32;
        when Exception_PACFail           ec = 0x1A; assert !from_32;
        when Exception_InstructionAbort  ec = 0x1E;    il_is_valid = FALSE;
        when Exception_PCallignment      ec = 0x20;    il_is_valid = FALSE;
        when Exception_DataAbort         ec = 0x21;
        when Exception_NV2DataAbort      ec = 0x22;
        when Exception_SPAlignment       ec = 0x26; assert !from_32;
        when Exception_FPTrapError       ec = 0x28;
        when Exception_SError            ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint       ec = 0x30;    il_is_valid = FALSE;
        when Exception_SoftwareStep      ec = 0x32;    il_is_valid = FALSE;
        when Exception_Watchpoint        ec = 0x34;    il_is_valid = FALSE;
        when Exception_NV2Watchpoint     ec = 0x35;
        when Exception_SoftwareBreakpoint ec = 0x38;
        when Exception_VectorCatch       ec = 0x3A; assert from_32;
        otherwise Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;
    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1'; // AArch64 instructions always 32-bit

    return (ec,il);
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;
    (ec, il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;
    iss2 = exception.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24, 0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = (Zeros(27) : // <63:37>
        iss2 : // <36:32>
        ec<5:0> : // <31:26>
        il : // <25>
        iss); // <24:0>

    if exceptype IN {Exception/InstructionAbort, Exception/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 IsSecureEL2Enabled() && IsSecure() then
                HPFAR_EL2.NS = exception.NS;
            else
                HPFAR_EL2.NS = '0';
            else
                HPFAR_EL2.NS = '0';
            end
            HPFAR_EL2<43:4> = bits(40) UNKNOWN;
        end

    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

    // 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);
AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)

exception = ExceptionSyndrome(Exception_FPTrapedException);

if is_ase then
    if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
        exception.syndrome<23> = '1';                          // TFV
    else
        exception.syndrome<23> = '0';                          // TFV
else
    exception.syndrome<23> = '1';                              // TFV

exception.syndrome<10:8> = bits(3) UNKNOWN;                    // VECITR

if exception.syndrome<23> == '1' then
    exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
else
    exception.syndrome<7,4:0> = bits(6) UNKNOWN;

route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

if PSTATE.EL > PSTATE(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
else if route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

AArch64.CallHypervisor(bits(16) immediate)

assert HaveEL(EL2);

if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();

bits(64) preferred_exception_return = NextInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_HypervisorCall);
exception.syndrome<15:0> = immediate;

if PSTATE.EL == EL3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

// AArch64.CallSupervisor()
// ========================

// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
AArch64.TakeException(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[target_el].IESB == '1';
if HaveDoubleFaultExt() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && target_el == EL3);
if sync_errors && InsertIESBBeforeException(target_el) then
    SynchronizeErrors();
    iesb_req = FALSE;
sync_errors = FALSE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero from 32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
MaybeZeroSVEUppers(target_el);

if UINT(target_el) > UINT(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    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);
    
elseif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;

bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState);

if PSTATE.EL == EL1 & target_el == EL1 & EL2Enabled() then
    if HaveNV2Ext() & (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
        spsr<3:2> = '10';
    else
        if HaveNVExt() & HCR_EL2.<NV,NV1> == '10' then
            spsr<3:2> = '10';
        if HaveBTIExt() & !UsingAArch32() then
            // SPSR[].BTYPE is only guaranteed valid for these exception types
            if exception.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                Exception_SoftwareStep, Exception_PCAlignment,
                Exception_InstructionAbort, Exception_Breakpoint,
                Exception_VectorCatch, Exception_SoftwareBreakpoint, Exception_IllegalState, Exception_BranchTarget} then
                zero_btype = FALSE;
            else
                zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
                if zero_btype then spsr<11:10> = '00';
            if HaveNV2Ext() & exception.exceptype == Exception_NV2DataAbort & target_el == EL3 then
                // external aborts are configured to be taken to EL3
                exception.exceptype = Exception_DataAbort;
                if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
                    AArch64_ReportException(exception, target_el);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';
SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SS = '0';
ShouldAdvanceSS = FALSE;
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from 32 then                             // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0';                         // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCTLR[].SPAN == '0') then
    PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].SSBS;
if HaveMTEExt() then PSTATE.TCO = '1';

BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

if sync_errors then
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

EndOfInstruction();

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

// AArch64.AArch32SystemAccessTrap()
// =================================
// Trapped AARCH32 system register access.
AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;
exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)

    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
      elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>; // opc1
        if instr<19:16> == '1111' then // Rt==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>;
        endif
        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<9:5> = bits(5) UNKNOWN;
          iss<3> = '1';
        else
          exception.exceptype == Exception_Uncategorized then
            // Trapped for unknown reason
            iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
            iss<3> = '0';
          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)
   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
      // Check if access disabled in CPACR_EL1
      case CPACR_EL1.FPEN of
         when 'x0' disabled = TRUE;
         when '01' disabled = PSTATE.EL == EL0;
         when '11' disabled = FALSE;
      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
        if HaveVirtHostExt() & HCR_EL2.E2H == '1' then
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 & HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
                if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
            else
                if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
        end
        if HaveEL(EL3) then
            if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
        end
    end
    return;

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETAA, ERATAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)
    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETAA, ERATAB 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
            exception.syndrome<1> = '0';
            exception.syndrome<0> = '0'; // RES0
        else
            exception.syndrome<1> = '1';
            if pac_uses_key_a then
                exception.syndrome<0> = '0'; // ERETAA
            else
                exception.syndrome<0> = '1';
        end
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    end
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
   if PSTATE.EL == EL0 then UNDEFINED;
   if (!PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1' then
       UNDEFINED;
   route_to_el2 = FALSE;
   if !HaveEL(EL3) then
      if PSTATE.EL == EL1 && EL2Enabled() then
         if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
            route_to_el2 = TRUE;
         else
            UNDEFINED;
      else
         UNDEFINED;
   else
      route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
   if route_to_el2 then
      bits(64) preferred_exception_return = ThisInstrAddr();
      vect_offset = 0x0;
      exception = ExceptionSyndrome(Exception_MonitorCall);
      exception.syndrome<15:0> = imm;
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

// AArch64.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
   if HaveFGTExt() then
      route_to_el2 = FALSE;
   if PSTATE.EL == EL0 then
      route_to_el2 = (!ELUsingAArch32(EL0) && !ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1')
                       && (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));
   elsif PSTATE.EL == EL1 then
      route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1')
                       && (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));
   if route_to_el2 then
      exception = ExceptionSyndrome(Exception_SupervisorCall);
      exception.syndrome<15:0> = immediate;
      bits(64) preferred_exception_return = ThisInstrAddr();
      vect_offset = 0x0;
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```c
// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
    if trap then
        AArch64.WFxTrap(target_el, is_wfe);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```c
// 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';
        preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_IllegalState);
        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```c
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_Uncategorized);
    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```
Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

// AArch64.SystemAccessTrap()
// =========================
// Trapped access to AArch64 system register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) & target_el != EL0 & UInt(target_el) >= UInt(PSTATE.EL);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

// AArch64.SystemAccessTrapSyndrome()
// ==============================================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;
    case ec of
        when 0x0                                                     // 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.
            exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
            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);
AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)

assert UInt(target_el) > UInt(PSTATE.EL);

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_WFxTrap);
exception.syndrome<24:20> = ConditionSyndrome();
exception.syndrome<0> = if is_wfe then '1' else '0';

if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

CheckFPAdvSIMDEnabled64()

boolean trap = FALSE;
bits(25) iss = ZeroExtend('10'); // 0x2

if PSTATE.EL == EL0 then
    if !IsInHost() then
        trap = SCTLR_EL1.EnALS == '0';
        target_el = if HCR_EL2.TGE == '1' then EL2 else EL1;
    else
        trap = SCTLR_EL2.EnALS == '0';
        target_el = EL2;

if !trap && ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1) then
    if trap && EL2Enabled() && HCRX_EL2.EnALS == '0';
    target_el = EL2;

if trap then LDST64BTrap(target_el, iss);
// CheckST64BV0Enabled()
// ====================
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
  boolean trap = FALSE;
  bits(25) iss = ZeroExtend('1'); // 0x1

  if PSTATE.EL == EL0 then
    if !IsInHost() then
      trap = SCTLR_EL1.EnAS0 == '0';
      target_el = if HCR_EL2.TGE == '1' then EL2 else EL1;
    else
      trap = SCTLR_EL2.EnAS0 == '0';
      target_el = EL2;
  if !trap && ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1) then
    trap = EL2Enabled() && HCRX_EL2.EnAS0 == '0';
    target_el = EL2;
  if !trap && ((PSTATE.EL IN {EL0, EL1}) || PSTATE.EL == EL2) then
    trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
    target_el = EL3;
  if trap then LDST64BTrap(target_el, iss);

// CheckST64BVEnabled()
// ====================
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
  boolean trap = FALSE;
  bits(25) iss = Zeros();

  if PSTATE.EL == EL0 then
    if !IsInHost() then
      trap = SCTLR_EL1.EnASR == '0';
      target_el = if HCR_EL2.TGE == '1' then EL2 else EL1;
    else
      trap = SCTLR_EL2.EnASR == '0';
      target_el = EL2;
  if !trap && ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1) then
    trap = EL2Enabled() && HCRX_EL2.EnASR == '0';
    target_el = EL2;
  if trap then LDST64BTrap(target_el, iss);

// LDST64BTrap()
// =============
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;

  exception = ExceptionSyndrome(Exception_LDST64BTrap);
  exception.syndrome = iss;
  AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
  return;
**Library pseudocode for aarch64/exceptions/traps/WFETrapDelay**

```plaintext
// WFETrapDelay()
// ==============
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTLR_EL1.TWEDEn == '1';
                delay = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2.TWEDEn == '1';
                delay = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
        when EL2
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay = 1 << (UInt(SCR_EL3.TWEDEL) + 8);
    return (delay_enabled, delay);
```

**Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay**

```plaintext
// WaitForEventUntilDelay()
// ========================
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay)
    boolean eventarrived = FALSE;
    // set eventarrived to TRUE if WaitForEvent() returns before
    // 'delay' expires when delay_enabled is TRUE.
    return eventarrived;
```

**Library pseudocode for aarch64/functions/aborts/AArch64.CreateFaultRecord**

```plaintext
// AArch64.CreateFaultRecord()
// ===========================

FaultRecord AArch64.CreateFaultRecord(Fault statuscode, bits(52) ipaddress, boolean NS,
                                        integer level, AccType acctype, boolean write, bit extflag,
                                        bits(2) errortype, boolean secondstage, boolean s2fs1walk)
    FaultRecord fault;
    fault.statuscode = statuscode;
    fault.domain = bits(4) UNKNOWN;  // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN; // Not used from AArch64
    fault.errortype = errortype;
    fault.ipaddress.NS = if NS then '1' else '0';
    fault.ipaddress.address = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;
    return fault;
```
AArch64.FaultSyndrome()

---

// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception Level using AArch64.

(bits(25), bits(5)) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;
    bits(25) iss = Zeros();
    bits(5) iss2 = Zeros();

    if (HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 &&
        fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
        iss2 = AArch64.RegisterSpecifier();
    if HaveRASExt() && IsAsyncAbort(fault) then
        if (HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 &&
            fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
            iss<12:11> = '01';  // LST
        else
            iss<12:11> = fault.errortype;  // SET
    if d_side then
        if (IsSecondStage(fault) && !fault.s2fs1walk && (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.acctype == AccType_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")) ) then
            iss<24:14> = LSInstructionSyndrome();
        if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
            iss<13> = '1';  // Fault is generated by use of VNCR EL2
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss, iss2);
```
boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.
acctype = 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);
    ClearExclusiveLocal(ProcessorID());
    if passed then
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
return passed;
```

**Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA**

```
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);

// 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.
bool AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

**Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA**

```
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

// 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/AArch64.SetExclusiveMonitors**

```plaintext
// AArch64.SetExclusiveMonitors()
// ==============================

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then return;
    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
        MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
        AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

**Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused**

```plaintext
// FPRSqrtStepFused()
// ==================

bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';                // Use RNE rounding mode
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elseif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);
        end
    else
        return result;
```

Shared Pseudocode Functions
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;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);

    boolean altpf = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altpf;               // Generate no floating-point exceptions
    if altpf then fpcr.<FIZ,FZ> = '11';   // Flush denormal input and output to zero
    if altpf then fpcr.RMode    = '00';   // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1  = (type1 == FPType_Infinity);
        inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);
            end
    end

    return result;
Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

// AArch64.AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccType acctype)
if PSTATE.M<4> == '1' then return FALSE;
if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
  return FALSE;
if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then
  return FALSE;
if !AArch64.AllocationTagAccessIsEnabled(acctype) then
  return FALSE;
if acctype IN {AccType_IFETCH, AccType_TTW} then
  return FALSE;
if acctype == AccType_NV2REGISTER then
  return FALSE;
if PSTATE.TC0=='1' then
  return FALSE;
if !IsTagCheckedInstruction() then
  return FALSE;
return TRUE;

Library pseudocode for aarch64/functions/memory/AArch64.AddressWithAllocationTag

// AArch64.AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, AccType acctype, bits(4) allocation_tag)
  bits(64) result = address;
  bits(4) tag;
  if AArch64.AllocationTagAccessIsEnabled(acctype) then
    tag = allocation_tag;
  else
    tag = '0000';
  result<59:56> = tag;
  return result;

Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

// AArch64.AllocationTagFromAddress()
// ==================================
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
  return tagged_address<59:56>;
#### Library pseudocode for aarch64/functions/memory/AArch64.CheckAlignment

```c
// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype, boolean iswrite)
{
    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC, 
                          AccType_ORDEREDATOMICRW, AccType_ATOMICLS64};
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMICRW, 
                          AccType_ORDEREDATOMIC, AccType_ATOMICLS64};
    vector = acctype == AccType_VEC;
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveLSE2Ext() then
        check = (UInt(address<0+:4>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
    else check = atomic || ordered;
    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
    return aligned;
}
```

#### Library pseudocode for aarch64/functions/memory/AArch64.CheckTag

```c
// 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 actype, boolean wasaligned]
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
bits(size*8) value;
iswrite = FALSE;

memaddrdesc = AArch64.TranslateAddress(address, actype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), actype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), actype, iswrite);
    value = _Mem[memaddrdesc, size, accdesc, FALSE];
return value;

// AArch64.MemSingle[] - assignment (write) form
// ================================================================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType actype, boolean wasaligned] = bits(size*8) value
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;

memaddrdesc = AArch64.TranslateAddress(address, actype, 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
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), actype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), actype, iswrite);
    _Mem[memaddrdesc, size, accdesc] = value;
return;
// AArch64.MemTag[] - non-assignment (read) form
// ---------------------------------------------------------------------------
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bhits(64) address, AccType acctype]
  assert acctype == AccType_NORMAL;
  AddressDescriptor memaddrdesc;
  bits(4) value;
  iswrite = FALSE;

  memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, TRUE, TAG_GRANULE);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch64.Abort(address, memaddrdesc.fault);

// Return the granule tag if tagging is enabled...
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
  return _MemTag[memaddrdesc];
else
  // ...otherwise read tag as zero.
  return '0000';

// AArch64.MemTag[] - assignment (write) form
// ---------------------------------------------------------------------------
// Store an Allocation Tag to memory.

AArch64.MemTag[bhits(64) address, AccType acctype] = bits(4) value
  assert acctype == AccType_NORMAL;
  AddressDescriptor memaddrdesc;
  iswrite = TRUE;
// Stores of allocation tags must be aligned
if address != Align(address, TAG_GRANULE) then
  boolean secondstage = FALSE;
  AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

wasaligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, TAG_GRANULE);

// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == MemType_Device then
  c = ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE);
  assert c IN {Constraint_NONE, Constraint_FAULT};
  if c == Constraint_FAULT then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
  _MemTag[memaddrdesc] = value;

Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

// AArch64.PhysicalTag()
// ---------------------
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bhits(64) vaddr)
  return vaddr<59:56>;
// AArch64.TranslateAddressForAtomicAccess()
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeInBits)
    boolean isWrite = FALSE;
    size = sizeInBits DIV 8;
    assert size IN {1, 2, 4, 8, 16};
    aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, isWrite);
    // MMU or MPU lookup
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, isWrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);
    if HaveMTEExt() && AArch64.AccessIsTagChecked(address, AccType_ATOMICRW) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, ptag, isWrite) then
            AArch64.TagCheckFault(address, AccType_ATOMICRW, isWrite);
    return memaddrdesc;

Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
boolean AddressSupportsLS64(bits(64) address);

Library pseudocode for aarch64/functions/memory/CheckSPAlignment

// CheckSPAlignment()
// ================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    if PSTATE.EL == EL0 then
        stack_align_check = (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();
// Returns True if the current instruction uses tag-checked memory access,
// False otherwise.

boolean IsTagCheckedInstruction();
// 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 BigEndian(acctype) then
      value = BigEndianReverse(value);
    return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
boolean iswrite = TRUE;
if BigEndian(acctype) then
  value = BigEndianReverse(value);

aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
  atomic = aligned;
else
  // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
  // 64-bit aligned.
  atomic = address == Align(address, 8);
if !atomic then
  assert size > 1;
  AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
**assert c IN {Constraint_FAULT, Constraint_NONE};**

**if c == Constraint_NONE then aligned = TRUE;**

**for i = 1 to size-1**

**AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;**

**elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then**

**AArch64.MemSingle[address, 8, acctype, aligned] = value<63:0>;**

**AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>;**

**else**

**AArch64.MemSingle[address, size, acctype, aligned] = value;**

**return;**

---

Library pseudocode for aarch64/functions/memory/MemAtomic

```plaintext
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType stacctype)

bits(size) newvalue;

memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);

ldaccdesc = CreateAccessDescriptor(ldacctype);

staccdesc = CreateAccessDescriptor(stacctype);

// All observers in the shareability domain observe the
// following load and store atomically.

oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc, FALSE];

if BigEndian(ldacctype) then
    oldvalue = BigEndianReverse(oldvalue);

case op of
    when MemAtomicOp_ADD newvalue = oldvalue + value;
    when MemAtomicOp_BIC newvalue = oldvalue AND NOT(value);
    when MemAtomicOp_EOR newvalue = oldvalue EOR value;
    when MemAtomicOp_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;

if BigEndian(stacctype) then
    newvalue = BigEndianReverse(newvalue);

_Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;

// Load operations return the old (pre-operation) value
return oldvalue;
```

---

Shared Pseudocode Functions Page 2588
Library pseudocode for aarch64/functions/memory/MemAtomicCompareAndSwap

// MemAtomicCompareAndSwap()
// =========================
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
    bits(size) newvalue, AccType ldacctype, AccType stacctype)

    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc, FALSE];
    if BigEndian(ldacctype) then
        oldvalue = BigEndianReverse(oldvalue);
    if oldvalue == expectedvalue then
        if BigEndian(stacctype) then
            newvalue = BigEndianReverse(newvalue);
            _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;
    return oldvalue;
Library pseudocode for aarch64/functions/memory/MemLoad64B

// MemLoad64B()
// ============
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccType acctype)
    assert address == Align(address, 64);
    bits(512) data;
    boolean iswrite = FALSE;
    boolean aligned = TRUE;
    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            fault = AArch64.ExclusiveFault(acctype, iswrite);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level
            for i = 0 to 63
                data<7+8*i : 8*i> = AArch64.MemSingle[address+8*i, 1, acctype, aligned];
            return data;
    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, 64);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTEExt() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
                AArch64.TagCheckFault(address, acctype, iswrite);
            data = _Mem[memaddrdesc, 64, accdesc, iswrite];
        return data;
Library pseudocode for aarch64/functions/memory/MemStore64B

```c
// MemStore64B()
// =============
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccType acctype)
    assert address == Align(address, 64);
    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            iswrite = TRUE;
            fault = AArch64.ExclusiveFault(acctype, iswrite);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level.
            aligned = TRUE;
            for i = 0 to 63
                AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
        else
            return MemStore64BWithRet(address, value, acctype);  // Return status is ignored by ST64B
    return;
```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

```c
// MemStore64BWithRet()
// ====================
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccType acctype)
    assert address == Align(address, 64);
    AddressDescriptor memaddrdesc;
    boolean iswrite = TRUE;
    boolean aligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, 64);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
        return ZeroExtend('1');
    
    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    
    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTEExt() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
                AArch64.TagCheckFault(address, acctype, iswrite);
                return ZeroExtend('1');
            
            _Mem[memaddrdesc, 64, accdesc] = value;
            status = MemStore64BWithRetStatus();
        return status;
    ```
Library pseudocode for aarch64/functions/memory/MemStore64BWithRetStatus

// Generates the return status of memory write with ST64BV or ST64BV0 instructions. The status indicates if the operation succeeded, failed, or was not supported at this memory location.
bits(64) MemStore64BWithRetStatus();

Library pseudocode for aarch64/functions/memory/NVMem

// NVMem[] - non-assignment form
// ----------------------------------------------
// This function is the load memory access for the transformed System register read access when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
assert offset > 0;
bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
return Mem[address, 8, AccType_NV2REGISTER];

// NVMem[] - assignment form
// ----------------------------------------------
// This function is the store memory access for the transformed System register write access when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
assert offset > 0;
bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
Mem[address, 8, AccType_NV2REGISTER] = value;
return;

Library pseudocode for aarch64/functions/memory/SetTagCheckedInstruction

// Flag the current instruction as using/not using memory tag checking.
SetTagCheckedInstruction(boolean checked);
// 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 = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
  integer top_bit = if tbi then 55 else 63;
  if tagged pointers are in use for a regime with two TTBRs, use bit<55> of
  // the pointer to select between upper and lower ranges, and preserve this.
  // This handles the awkward case where there is apparently no correct choice between
  // the upper and lower address range - ie an addr of 1xxxxxxxxx0... with TBI0=0 and TBI1=1
  // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
  if PtrHasUpperAndLowerAddRanges() then
    assert SITranslationRegime() IN {EL1, EL2};
    if SITranslationRegime() == 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>;
        end if;
      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>;
        end if;
      end if;
    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>;
        end if;
      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>;
        end if;
      end if;
    end if;

    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);
    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);
    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
      ext_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
      ext_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
    end if;
    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 if;
  end if;

  PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

  // 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;
else
    return AddPAC(X, Y, APDAKey_EL1, TRUE);

Library pseudocode for aarch64/functions/pac/addpaca/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 = SITTranslationRegime() == 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;
elseif TrapEL2 then TrapPACUse(EL2);
elseif TrapEL3 then TrapPACUse(EL3);
else return AddPAC(X, Y, APDBKey_EL1, TRUE);
Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

```c
// 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 = FALSE;
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    endcase

    if TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>)<63:32>:Zeros(32);
```

Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

// 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 = $iTranslationRegime() == 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);
Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```plaintext
// AddPACIB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = SITranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end case
    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

```plaintext
// 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 = ExceptionSyndrome(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);
```
Library pseudocode for aarch64/functions/pac/auth/Auth

// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number, boolean is_combined)

bits(64) PAC;
bits(64) result;
bits(64) original_ptr;
bits(2) error_code;
bits(64) extfield;

// Reconstruct the extension field used of adding the PAC to the pointer
boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
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<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>;
  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
    result = ptr;
  return result;
else
  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<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
    result<63:56> = result<63:56> EOR PAC<63:56>;
    if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
      if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
        error_code = (if data then '1' else '0'):key_number;
        AArch64.PACFailException(error_code);
  return result;
Library pseudocode for aarch64/functions/pac/authda/AuthDA

// AuthDA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y, boolean is_combined)
boolean TrapEL2;
boolean TrapEL3;
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 = SITranslationRegime() == 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);
Library pseudocode for aarch64/functions/pac/authdb/AuthDB

// AuthDB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y, boolean is_combined)

boolean TrapEL2;
boolean TrapEL3;

bits(1) Enable;
bits(128) APDBKey_EL1;

APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
      (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnDB;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3/API == '0';
  when EL2
    Enable = SCTLR_EL2.EnDB;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnDB;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return Auth(X, Y, APDBKey_EL1, TRUE, '1', is_combined);

 Shared Pseudocode Functions
Library pseudocode for aarch64/functions/pac/authia/AuthIA

// 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

```plaintext
// 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;
    endcase
    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);
```

Shared Pseudocode Functions
integer CalculateBottomPACBit(bit top_bit)
    integer tsz_field;
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if top_bit == '1' then
                tsz_field = UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
                using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
            else
                // EL2 translation regime registers
                assert HaveEL2();
                tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
                using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
            
            max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if using64k then 47 else 48);
            if tsz_field > max_limit_tsz_field then
                // TCR_ELx.TySZ is out of range
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_NONE};
                if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
                tszmin = if using64k & VAMax() == 52 then 12 else 16;
                if tsz_field < tszmin then
                    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                    assert c IN {Constraint_FORCE, Constraint_NONE};
                    if c == Constraint_FORCE then tsz_field = tszmin;
                return (64-tsz_field);
        
    else
        tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
        using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';
        max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if using64k then 47 else 48);
        if tsz_field > max_limit_tsz_field then
            // TCR_ELx.TySZ is out of range
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_NONE};
            if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
            tszmin = if using64k & VAMax() == 52 then 12 else 16;
            if tsz_field < tszmin then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_NONE};
                if c == Constraint_FORCE then tsz_field = tszmin;
            return (64-tsz_field);
Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```c
array (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] = 0x13198A2E03707344<63:0>;
  RC[2] = 0xA4093822299F31D0<63:0>;
  RC[3] = 0x082EFA98EC4EC89<63:0>;
  RC[4] = 0x452821E638B01377<63:0>;
  modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
  runningmod = modifier;
  workingval = data EOR key0;
  for i = 0 to 4
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = workingval EOR RC[i];
    if i > 0 then
      workingval = PACCellShuffle(workingval);
      workingval = PACMult(workingval);
      runningmod = PACSub(runningmod);
      roundkey = modk0 EOR runningmod;
      workingval = workingval EOR roundkey;
      workingval = PACCellShuffle(workingval);
      workingval = PACMult(workingval);
      workingval = PACCellShuffle(workingval);
      workingval = PACMult(workingval);
      workingval = PACCellShuffle(workingval);
      workingval = PACMult(workingval);
      workingval = key1 EOR workingval;
      workingval = PACCellShuffle(workingval);
      workingval = PACInvSub(workingval);
      workingval = PACMult(workingval);
      workingval = PACCellShuffle(workingval);
      workingval = PACMult(workingval);
      workingval = PACCellShuffle(workingval);
      workingval = workingval EOR key0;
    end
    for i = 0 to 4
      workingval = PACInvSub(workingval);
      if i < 4 then
        workingval = PACMult(workingval);
        workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR RC[4-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
      end
      workingval = workingval EOR modk0;
    end
  end
  return workingval;
}```
Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```c
// 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

```c
// 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<43:40>;
    outdata<59:56> = indata<63:60>;
    return outdata;
```
PACInvSub()

This is a 4-bit substitution from the PRINCE-family cipher

for i = 0 to 15
    case Tinput<4*i+3:4*i> of
    when '0000'  Toutput<4*i+3:4*i> = '0101';
    when '0001'  Toutput<4*i+3:4*i> = '1110';
    when '0010'  Toutput<4*i+3:4*i> = '1101';
    when '0011'  Toutput<4*i+3:4*i> = '1000';
    when '0100'  Toutput<4*i+3:4*i> = '1010';
    when '0101'  Toutput<4*i+3:4*i> = '1011';
    when '0110'  Toutput<4*i+3:4*i> = '0001';
    when '0111'  Toutput<4*i+3:4*i> = '1001';
    when '1000'  Toutput<4*i+3:4*i> = '0010';
    when '1001'  Toutput<4*i+3:4*i> = '0110';
    when '1010'  Toutput<4*i+3:4*i> = '1111';
    when '1011'  Toutput<4*i+3:4*i> = '0000';
    when '1100'  Toutput<4*i+3:4*i> = '0100';
    when '1101'  Toutput<4*i+3:4*i> = '1100';
    when '1110'  Toutput<4*i+3:4*i> = '0111';
    when '1111'  Toutput<4*i+3:4*i> = '0011';
    return Toutput;

PACMult()

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+16)+3:4*(i+16)>, 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+20)+3:4*(i+20)>, 1) EOR RotCell(Sinput<4*(i)+8+3:4*(i+8)>, 2);
    t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i)+4+3:4*(i+4)>, 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

```plaintext
// 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

```plaintext
// 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

```plaintext
// 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

```plaintext
// 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

```c
// 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<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
}
```

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```c
// 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<15:12>;
    outdata<27:24> = TweakCellRot(indata<35:32>);
    outdata<31:28> = TweakCellRot(indata<51:48>);
    outdata<35:32> = TweakCellRot(indata<55:52>);
    outdata<39:36> = TweakCellRot(indata<63:60>);
    outdata<43:40> = TweakCellRot(indata<47:44>);
    outdata<47:44> = TweakCellRot(indata<19:16>);
    outdata<51:48> = TweakCellRot(indata<35:32>);
    outdata<55:52> = TweakCellRot(indata<7:4>);
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
}
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC

```c
// 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

```c
// HaveEnhancedPAC2()
// =========

// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
{
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality");
}
```
Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

```plaintext
// HaveFPAC()
// =========
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return HaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPACCombined

```plaintext
// HaveFPACCombined()
// ==================
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return HaveFPAC() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```plaintext
// HavePACExt()
// ===========
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```plaintext
// PtrHasUpperAndLowerAddRanges()
// ==============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    return PSTATE.EL == EL1 || PSTATE.EL == EL0 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
```

Library pseudocode for aarch64/functions/pac/strip/Strip

```plaintext
// Strip()
// ========
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);
    if tbi then
        original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    return original_ptr;
```
Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.
TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64
AArch64.ESBOperation()
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
        PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));

    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSError Syndrome();
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSError Syndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
            ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0

        return;

    else

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSError Syndrome

// Return the SError syndrome
bits(25) AArch64.PhysicalSError Syndrome(boolean implicit_esb);
Library pseudocode for aarch64/functions/ras/AArch64.ReportDeferredSError

```c
// 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

```c
// 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>);
            HCR_EL2.VSE = '0';  // Clear pending virtual SError
        return;
```

Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```c
// 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;
    end

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros();
        return;
```

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```c
// AArch64.ResetGeneralRegisters()
// ------------------------------
AArch64.ResetGeneralRegisters()
  for i = 0 to 30
      X[i] = bits(64) UNKNOWN;
  return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```c
// AArch64.ResetSIMDFPRegisters()
// -----------------------------
AArch64.ResetSIMDFPRegisters()
  for i = 0 to 31
      V[i] = bits(128) UNKNOWN;
  return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```c
// AArch64.ResetSpecialRegisters()
// -------------------------------
AArch64.ResetSpecialRegisters()

  // AArch64 special registers
  SP_EL0 = bits(64) UNKNOWN;
  SP_EL1 = bits(64) UNKNOWN;
  SPSR_EL1 = bits(64) UNKNOWN;
  ELR_EL1 = bits(64) UNKNOWN;
  if HaveEL(EL2) then
      SP_EL2 = bits(64) UNKNOWN;
      SPSR_EL2 = bits(64) UNKNOWN;
      ELR_EL2 = bits(64) UNKNOWN;
  if HaveEL(EL3) then
      SP_EL3 = bits(64) UNKNOWN;
      SPSR_EL3 = bits(64) UNKNOWN;
      ELR_EL3 = bits(64) UNKNOWN;

  // AArch32 special registers that are not architecturally mapped to AArch64 registers
  if HaveAArch32EL(EL1) then
      SPSR_fiq<31:0> = bits(32) UNKNOWN;
      SPSR_irq<31:0> = bits(32) UNKNOWN;
      SPSR_abt<31:0> = bits(32) UNKNOWN;
      SPSR_und<31:0> = bits(32) UNKNOWN;

  // External debug special registers
  DLR_EL0 = bits(64) UNKNOWN;
  DSPSR_EL0 = bits(64) UNKNOWN;

  return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```c
AArch64.ResetSystemRegisters(boolean cold_reset);
```
Library pseudocode for aarch64/functions/registers/PC

// PC - non-assignment form
// ================
// Read program counter.

bits(64) PC[]
return _PC;

Library pseudocode for aarch64/functions/registers/SP

// SP[] - assignment form
// ===============
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
assert width IN {32,64};
if PSTATE.SP == '0' then
  SP_EL0 = ZeroExtend(value);
else
  case PSTATE.EL of
  when EL0 SP_EL0 = ZeroExtend(value);
  when EL1 SP_EL1 = ZeroExtend(value);
  when EL2 SP_EL2 = ZeroExtend(value);
  when EL3 SP_EL3 = ZeroExtend(value);
  return;

// 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>;

Library pseudocode for aarch64/functions/registers/V

// V[] - assignment form
// =============
// Write to SIMD&FP register with implicit extension from 8, 16, 32 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;
        _R[n] = Zeros(width);
        return;
    else
        assert width == 64;
        bits(64) vreg = V[n];
        _R[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;
    else
        _R[31] = ZeroExtend(value);
        return;

// X[] - non-assignment form
// =========================
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<<width-1:0>;
    else
        return Zeros(width);
// AArch32.IsFPEnabled()
// ===============
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
if el == EL0 & !ELUsingAArch32(EL1) then
    return AArch64.IsFPEnabled(el);

if HaveEL(EL3) & !ELUsingAArch32(EL3) & !IsSecure() then
    // Check if access disabled in NSACR
    if NSACR.cp10 == '0' then return FALSE;

if el IN {EL0, EL1} then
    // Check if access disabled in CPACR
    case CPACR.cp10 of
    when '00' disabled = TRUE;
    when '01' disabled = el == EL0;
    when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
    when '11' disabled = FALSE;
    if disabled then return FALSE;

if el IN {EL0, EL1, EL2} & !EL2Enabled() then
    if !ELUsingAArch32(EL2) then
        return AArch64.IsFPEnabled(EL2);
    if HCPTR.TCP10 == '1' then return FALSE;

if HaveEL(EL3) & !ELUsingAArch32(EL3) then
    // Check if access disabled in CPTR_EL3
    if CPTR_EL3.TFP == '1' then return FALSE;

return TRUE;

// AArch64.IsFPEnabled()
// ===============
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
// Check if access disabled in CPACR_EL1
if el IN {EL0, EL1} & !IsInHost() then
    // Check FP&SIMD at EL0/EL1
    case CPACR_EL1.FPEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = el == EL0;
    when '11' disabled = FALSE;
    if disabled then return FALSE;

// Check if access disabled in CPTR_EL2
if el IN {EL0, EL1, EL2} & !EL2Enabled() then
    if HaveVirtHostExt() & HCR_EL2.E2H == '1' then
        case CPTR_EL2.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = el == EL0 & HCR_EL2.TGE == '1';
        when '11' disabled = FALSE;
        if disabled then return FALSE;
    else
        if CPTR_EL2.TFP == '1' then return FALSE;

// Check if access disabled in CPTR_EL3
if HaveEL(EL3) then
    if CPTR_EL3.TFP == '1' then return FALSE;

return TRUE;
Library pseudocode for aarch64/functions/sve/CeilPow2

```c
// 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

```c
// CheckSVEEnabled()
// ================
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);

    case CPACR_EL1.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);
```

// Check if access disabled in CPACR_ELO
if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
    case CPACR_ELO.ZEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    if disabled then SVEAccessTrap(EL0);

    case CPACR_ELO.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL0);

    case CPACR_ELO.TZ of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    if disabled then SVEAccessTrap(EL0);

    else
        if CPTR_ELO.TZ == '1' then SVEAccessTrap(EL0);
        if CPTR_ELO.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL0);
```

// Check if access disabled in CPACR_EL1
if PSTATE.EL IN {EL0, EL1, EL2} && !EL2Enabled() then
    if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
        case CPTR_EL2.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL2);

    case CPTR_EL2.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
        when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
```

// Check if access disabled in CPTR_EL2
if PSTATE.EL IN {EL0, EL1, EL2} && !EL2Enabled() then
    if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
        case CPTR_EL2.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL2);

    case CPTR_EL2.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
        when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
```

// 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

```c
// DecodePredCount()
// ================

integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = VLDIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise    numElem = 0;
    return numElem;
```

Library pseudocode for aarch64/functions/sve/ElemFFR

```c
// 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

```c
// 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_SVEZERUOPPER) then
        _FFR = ZeroExtend(value);
    else
        _FFR<width-1:0> = value;

Library pseudocode for aarch64/functions/sve/FPCompareNE

// FPCompareNE()
// =============

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    if op1_nan || op2_nan then
        result = TRUE;
    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExclusiveInvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
    FPProcessDenorms(type1, type2, N, fpcr);
    return result;

Library pseudocode for aarch64/functions/sve/FPCompareUN

// FPCompareUN()
// =============

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExclusiveInvalidOp, fpcr);
    result = type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN};
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```c
// FPConvertSVE()
// ==============
bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding);
// FPConvertSVE()
// ==============
bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```
// 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 = 0x0ea43a;
    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 = 0x3d08a4;
when 37 result = 0x3f179a;
when 38 result = 0x412c4d;
when 39 result = 0x4346cd;
when 40 result = 0x45672a;
when 41 result = 0x478d75;
when 42 result = 0x49b9be;
when 43 result = 0x4bec15;
when 44 result = 0x4e248c;
when 45 result = 0x506334;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599d16;
when 50 result = 0x5bfbb8;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 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 = 0x0e3ec32d3d1a2;
    when 6 result = 0x11301d0125b51;
    when 7 result = 0x1429aae92de9;
    when 8 result = 0x172b83c7d517b;
    when 9 result = 0x1a35beb6fc875;
    when 10 result = 0x1d4873168b99a;
    when 11 result = 0x206388628cd6;
    when 12 result = 0x2387a6e756238;
    when 13 result = 0x26b456e527cd0;
    when 14 result = 0x29e9df5f1fdee1;
    when 15 result = 0x2d285a6e4030b;
    when 16 result = 0x306fe0a31b715;
    when 17 result = 0x33c0bb26416ff;
    when 18 result = 0x371a7373aa9cb;
    when 19 result = 0x3a7db34e59ff7;
    when 20 result = 0x3dea64c123422;
    when 21 result = 0x4160a21f7f2e2a;
    when 22 result = 0x44e0860618920;
    when 23 result = 0x486a2b5c13cd0;
    when 24 result = 0x4bfada5362aa27;
    when 25 result = 0x4f9b2769d2ca7;
    when 26 result = 0x5342b569d0f82;
    when 27 result = 0x56f4736b527da;
    when 28 result = 0x5ab07dd485429;
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 = 0x75FEB564267C9;
when 36 result = 0x7A11473E0B187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x825899944CE13;
when 39 result = 0x868D99B4492ED;
when 40 result = 0x8ACE5422A00DB;
when 41 result = 0x8F1AE99157736;
when 42 result = 0x93737B06C0C5E5;
when 43 result = 0x970829FDE4E50;
when 44 result = 0x9C49182A3F090;
when 45 result = 0xA0C667B5DE565;
when 46 result = 0xA5508323E255D;
when 47 result = 0xA9E6B5579FDFB;
when 48 result = 0xAE89F995AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F76F2F5E47;
when 51 result = 0xBC1E904BC1D2;
when 52 result = 0xC199BDD85529C;
when 53 result = 0xC67F1257D14B;
when 54 result = 0xCB720DCEF9069;
when 55 result = 0xD072D4A07897C;
when 56 result = 0xD581B0CF8A487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDF97337B9B5F;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEA4AFA2A4900A;
when 61 result = 0xEF01BEE615A27;
when 62 result = 0xF50765B6E454B;
when 63 result = 0xFA7C1819E900B;

return result<N-1:0>;

Library pseudocode for aarch64/functions/sve/FPMinNormal

// FPMinNormal()
// =============
bits(N) FPMinNormal(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Zeros(E-1)':'1';
frac = Zeros(F);
return sign : exp : frac;

Library pseudocode for aarch64/functions/sve/FPOne

// FPOne()
// ========
bits(N) FPOne(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-1);
frac = Zeros(F);
return sign : exp : frac;
Library pseudocode for aarch64/functions/sve/FPPointFive

```cpp
// 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

```cpp
// FPProcess()
// ===========

bits(N) FPProcess(bits(N) input)
assert N IN {16,32,64};
FPCRType fpcr = FPCR[];
(fptype,sign,value) = FPUnpack(input, fpcr);
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
    result = FPProcessNaN(fptype, input, fpcr);
elsif fptype == FPTYPE_Infinity then
    result = FPInfinity(sign);
elsif fptype == FPTYPE_Zero then
    result = FPZero(sign);
else
    result = FPRound(value, fpcr);
    FPProcessDenorm(fptype, N, fpcr);
return result;
```

Library pseudocode for aarch64/functions/sve/FPScale

```cpp
// 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);
    FPProcessDenorm(fptype, N, 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;
  coeff = FPTrigMAddCoefficient[x];
  boolean altfp = HaveAltFP() && fpcr.AH == '1';
  if altfp then
    (fptype,-,-) = FPUnpack(op2, fpcr, FALSE);
    if !(fptype IN {FPType_SNaN, FPType_QNaN}) then
      op2<N-1> = '0';
    else
      op2<N-1> = '0';
  result = FPMulAdd(coeff, op1, op2, fpcr);
  return result;
Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

// FPTrigMAddCoefficient()
// =======================

bits(N) FPTrigMAddCoefficient[integer index]
    assert N IN {16,32,64};
    integer result;
    if N == 16 then
        case index of
            when 0 result = 0x3c00;
            when 1 result = 0xb155;
            when 2 result = 0x2030;
            when 3 result = 0x0000;
            when 4 result = 0x0000;
            when 5 result = 0x0000;
            when 6 result = 0x0000;
            when 7 result = 0x0000;
            when 8 result = 0x3c00;
            when 9 result = 0xb800;
            when 10 result = 0x293a;
            when 11 result = 0x0000;
            when 12 result = 0x0000;
            when 13 result = 0x0000;
            when 14 result = 0x0000;
            when 15 result = 0x0000;
        elsif N == 32 then
            case index of
                when 0 result = 0x3f800000;
                when 1 result = 0xbe2aaaab;
                when 2 result = 0x3c088886;
                when 3 result = 0xb95008b9;
                when 4 result = 0x36369d6d;
                when 5 result = 0x00000000;
                when 6 result = 0x00000000;
                when 7 result = 0x00000000;
                when 8 result = 0x3f800000;
                when 9 result = 0xbf000000;
                when 10 result = 0x3d2aaaa6;
                when 11 result = 0xbab60705;
                when 12 result = 0x37cd37cc;
                when 13 result = 0x00000000;
                when 14 result = 0x00000000;
                when 15 result = 0x00000000;
            else // N == 64
                case index of
                    when 0 result = 0x3ff0000000000000;
                    when 1 result = 0xbfc555555555543;
                    when 2 result = 0x3f8111111110f30c;
                    when 3 result = 0xbf2a01a019b92fc6;
                    when 4 result = 0x3ec71de351f3d22b;
                    when 5 result = 0xbe5ae5e2b6f7b91;
                    when 6 result = 0x3de5d8408868552f;
                    when 7 result = 0x0000000000000000;
                    when 8 result = 0x3ff0000000000000;
                    when 9 result = 0xbf00000000000000;
                    when 10 result = 0x3fa555555555536;
                    when 11 result = 0xbf56c16c16c13a0b;
                    when 12 result = 0x3efa01a019b1e8d8;
                    when 13 result = 0xbe927e4f7282f468;
                    when 14 result = 0x3e21ee96d2641b13;
                    when 15 result = 0xbdba8f76380fb401;
            return result<N-1:0>;
Library pseudocode for aarch64/functions/sve/FPTrigSMul

// FPTrigSMul()
// ============
bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRTypre fpcr)
  assert N IN {16,32,64};
  result = FPMul(op1, op1, fpcr);
  (fptype, sign, value) = FPUnpack(result, fpcr);
  if (fptype != FPTypre_QNaN) && (fptype != FPTypre_SNaN) then
    result<N-1> = op2<0>;
    FPProcessDenorm(fptype, N, fpcr);
  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";
Library pseudocode for aarch64/functions/sve/HaveSVEFP32MatMulExt

// HaveSVEFP32MatMulExt()
// ======================
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP32 Matrix Multiply extension";

Library pseudocode for aarch64/functions/sve/HaveSVEFP64MatMulExt

// HaveSVEFP64MatMulExt()
// ======================
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP64MatMulExt()
return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP64 Matrix Multiply extension";

Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (e.g. power of two)

integer ImplementedSVEVectorLength(integer nbits)
return integer IMPLEMENTATION_DEFINED;

Library pseudocode for aarch64/functions/sve/IsEven

// IsEven()
// ========

boolean IsEven(integer val)
return val MOD 2 == 0;

Library pseudocode for aarch64/functions/sve/IsFPEnabled

// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point registers are enabled at the target exception level in the current execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
if ELUsingAArch32(el) then
    return AArch32.IsFPEnabled(el);
else
    return AArch64.IsFPEnabled(el);
// IsSVEEnabled()
// ==============
// Returns TRUE if access to SVE instructions and System registers is
// enabled at the target exception level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
if ELUsingAArch32(el) then
    return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

        // Check if access disabled in CPACR_EL2
        if el IN {EL0, EL1, EL2} && EL2Enabled() then
            if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
                case CPTR_EL2.ZEN of
                    when 'x0' disabled = TRUE;
                    when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                    when '11' disabled = FALSE;
                if disabled then return FALSE;
            else
                if CPTR_EL2.TZ == '1' then return FALSE;

        // Check if access disabled in CPTR_EL3
        if HaveEL(EL3) then
            if CPTR_EL3.EZ == '0' then return FALSE;
        return TRUE;

Library pseudocode for aarch64/functions/sve/LastActive

// 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';

Library pseudocode for aarch64/functions/sve/LastActiveElement

// 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;

Library pseudocode for aarch64/functions/sve/MAX_PL

constant integer MAX_PL = 256;

Library pseudocode for aarch64/functions/sve/MAX_VL

constant integer MAX_VL = 2048;
// MaybeZeroSVEUppers()
// ====================

MaybeZeroSVEUppers(bits(2) target_el)
   boolean lower_enabled;
   
   if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
      return;
   
   if target_el == EL3 then
      if EL2Enabled() then
         lower_enabled = IsFPEnabled(EL2);
      else
         lower_enabled = IsFPEnabled(EL1);
   elseif target_el == EL2 then
      assert !ELUsingAArch32(EL2);
      if HCR_EL2.TGE == '0' then
         lower_enabled = IsFPEnabled(EL1);
      else
         lower_enabled = IsFPEnabled(EL0);
   else
      assert target_el == EL1 && !ELUsingAArch32(EL1);
      lower_enabled = IsFPEnabled(EL0);
   
   if lower_enabled then
      integer vl = if IsSVEEnabled(PSTATE.EL) then VL else 128;
      integer pl = vl DIV 8;
      for n = 0 to 31
         if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _Z[n] = ZeroExtend(_Z[n]<vl-1:0>);
      for n = 0 to 15
         if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _P[n] = ZeroExtend(_P[n]<pl-1:0>);
      if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
         _FFR = ZeroExtend(_FFR<pl-1:0>);
// MemNF[] - non-assignment form
// =============================

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccType acctype]

assert size IN {1, 2, 4, 8, 16};
bits(8*size) value;

aligned = (address == Align(address, size));
A = SCTLR[].A;

if !aligned && (A == '1') then
  return (bits(8*size) UNKNOWN, TRUE);

atomic = aligned || size == 1;

if !atomic then
  (value<7:0>, bad) = MemSingleNF[address, 1, acctype, aligned];

  if bad then
    return (bits(8*size) UNKNOWN, TRUE);

  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;

    for i = 1 to size-1
      (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];

      if bad then
        return (bits(8*size) UNKNOWN, TRUE);
    else
      (value, bad) = MemSingleNF[address, size, acctype, aligned];
      if bad then
        return (bits(8*size) UNKNOWN, TRUE);

      if BigEndian(acctype) then
        value = BigEndianReverse(value);

  return (value, FALSE);
Library pseudocode for aarch64/functions/sve/MemSingleNF

// MemSingleNF[] - non-assignment form
// -------------------------------

(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, AccType acctype, boolean wasaligned]
  bits(8*size) value;
  boolean iswrite = FALSE;
  AddressDescriptor memaddrdesc;

  // Implementation may suppress NF load for any reason
  if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
    return (bits(8*size) UNKNOWN, TRUE);

  // MMU or MPU
  memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

  // Non-fault load from Device memory must not be performed externally
  if memaddrdesc.memattrs.memtype == MemType_Device then
    return (bits(8*size) UNKNOWN, TRUE);

  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    return (bits(8*size) UNKNOWN, TRUE);

  // Memory array access
  accdesc = CreateAccessDescriptor(acctype);
  if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(address, acctype) then
      bits(4) ptag = AArch64.PhysicalTag(address);
      if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
        return (bits(8*size) UNKNOWN, TRUE);

  value = _Mem[memaddrdesc, size, accdesc, iswrite];
  return (value, FALSE);

Library pseudocode for aarch64/functions/sve/NoneActive

// NoneActive()
// ============

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
  integer elements = N DIV (esize DIV 8);
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
  return '1';

Library pseudocode for aarch64/functions/sve/P

// P[] - non-assignment form
// --------------------------

bits(width) P[integer n]
  assert n >= 0 && n <= 31;
  assert width == PL;
  return _P[n]<width-1:0>;

// P[] - assignment form
// ----------------------

P[integer n] = bits(width) value
  assert n >= 0 && n <= 31;
  assert width == PL;
  if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
    _P[n] = ZeroExtend(value);
  else
    _P[n]<width-1:0> = value;
Library pseudocode for aarch64/functions/sve/PL

// PL - non-assignment form
// ========================

integer PL
    return VL DIV 8;

Library pseudocode for aarch64/functions/sve/PredTest

// PredTest()
// ===========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;

Library pseudocode for aarch64/functions/sve/ReducePredicated

// ReducePredicated()
// =================

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;

    for e = 0 to elements-1
        if e * esize < N & Elem[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

```c
// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    exception = Exception Syndrome(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

```c
enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```c
// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'fffffffffffffffffy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then return FALSE;
        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then return FALSE;
        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<15:8> == imm<7:0>) then return FALSE;
    
    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffffffy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then return FALSE;
        // Check for 'ffffffxy00xffffffffy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then return FALSE;
        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then return FALSE;

    return TRUE;
```
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);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));
    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);
    return vl;

Library pseudocode for aarch64/functions/sve/Z

// Z[] - non-assignment form
// =========================

bits(width) Z[integer n]
    assert n >= 0 && n <= 31;
    assert width == VL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====================

Z[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == VL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<width-1:0> = value;

Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

// CNTKCTL[] - non-assignment form
// ===============================

CNTKCTLType CNTKCTL[]
    bits(64) r;
    if IsInHost() then
        r = 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
// =============

CPACRType CPACR[]
    bits(64) r;
    if IsInHost() then
        r = CPTR_EL2;
        return r;
    r = CPACR_EL1;
    return r;

Library pseudocode for aarch64/functions/sysregisters/CPACRType

type CPACRType;

Library pseudocode for aarch64/functions/sysregisters/ELR

// ELR[] - non-assignment form
// ===========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
        return r;

// ELR[] - non-assignment form
// ===========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        when EL3  ELR_EL3 = r;
        otherwise Unreachable();
        return;

// ELR[] - assignment form
// =======================

ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
// ESR[] - non-assignment form
// ===========================
ESRType ESR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise unreachable();
    return r;

// ESR[] - assignment form
// =======================
ESR[bits(2) regime] = ESRType value
    bits(64) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise unreachable();
    return;

// ESR[] - assignment form
// =======================
ESR[] = ESRType value
    ESR[SITranslationRegime()] = value;

Library pseudocode for aarch64/functions/sysregisters/ESRTypetype ESRType;
Library pseudocode for aarch64/functions/sysregisters/FAR

```plaintext
// FAR[] - non-assignment form
// -----------------------------

bits(64) FAR[bites(2) regime]
    bits(64) r;
    case regime of
        when EL1    r = FAR_EL1;
        when EL2    r = FAR_EL2;
        when EL3    r = FAR_EL3;
        otherwise   Unreachable();
    return r;

// FAR[] - non-assignment form
// -----------------------------

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// ------------------------

FAR[bites(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1    FAR_EL1 = r;
        when EL2    FAR_EL2 = r;
        when EL3    FAR_EL3 = r;
        otherwise   Unreachable();
    return;

// FAR[] - assignment form
// ------------------------

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;

Library pseudocode for aarch64/functions/sysregisters/MAIR

```plaintext

// MAIR[] - non-assignment form
// -----------------------------

MAIRType MAIR[bites(2) regime]
    bits(64) r;
    case regime of
        when EL1    r = MAIR_EL1;
        when EL2    r = MAIR_EL2;
        when EL3    r = MAIR_EL3;
        otherwise   Unreachable();
    return r;

// MAIR[] - non-assignment form
// -----------------------------

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];

Library pseudocode for aarch64/functions/sysregisters/MAIRType

```plaintext

type MAIRType;
```
Library pseudocode for aarch64/functions/sysregisters/SCTLR

```c
// 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

```c
// VBAR[] - non-assignment form
// =============================

bits(64) VBAR[bits(2) regime] bits(64) r;
    case regime of
    when EL1  r = VBAR_EL1;
    when EL2  r = VBAR_EL2;
    when EL3  r = VBAR_EL3;
    otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// =============================

bits(64) VBAR[] return VBAR[S1TranslationRegime()];
```
Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// ======================================
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(AccType accType)
    bits(2) el = AArch64.AccessUsesEL(accType);
    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && el IN {EL0, EL1, EL2} && HCR_EL2.<E2H,TGE> != '11' then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && el == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && el == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && el == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && el == EL0 && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && el == EL0 && !(!EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') then
        return FALSE;
    else
        return TRUE;
```

Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// ===========================

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
                          bits(4) crm, bits(3) op2, bits(5) rt, bit read)
    return;
```

Library pseudocode for aarch64/functions/system/AArch64.ChooseNonExcludedTag

```
// AArch64.ChooseNonExcludedTag()
// ==============================
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag, bits(4) offset, bits(16) exclude)
    if IsOnes(exclude) then
        return '0000';
    if offset == '0000' then
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';
    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';
    return tag;
```
Library pseudocode for aarch64/functions/system/AArch64.ExecutingATS1xPIInstr

// AArch64.ExecutingATS1xPIInstr()
// ===========================
// Return TRUE if current instruction is AT S1E1R/WP

boolean AArch64.ExecutingATS1xPIInstr()
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> == '1101011010000000';
// 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;

// 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;

// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);

// Execute a system instruction with read (result operand).
// Returns the result of the instruction.
bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);

// 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);

// Write to a system register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);

// BTypeCompatible
boolean BTypeCompatible;
Library pseudocode for aarch64/functions/system/BTypeCompatible_BTIBoolean 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

Boolean BTypeCompatible_PACIXSP()
  if PSTATE.BTYPE IN {'01', '10'} then
    return TRUE;
  elseif 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/ChooseRandomNonExcludedTag

Bits(4) ChooseRandomNonExcludedTag(bits(16) exclude);

Library pseudocode for aarch64/functions/system/InGuardedPage

Boolean InGuardedPage;

Library pseudocode for aarch64/functions/system/SetBTypeCompatible

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/instrs/branch/eret/AArch64.ExceptionReturn

```c
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc, bits(64) spsr)

    SynchronizeContext();

    sync_errors = HaveIESB() &\& SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' &\& SCR_EL3.NMEA == '1' &\& PSTATE.EL == EL3);
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
   SendEventLocal();

    if PSTATE.IL == '1' &\& spsr<4> == '1' &\& spsr<20> == '0' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                  // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
        else                                     // Return to AArch64
            // ELR_ELx[63:56] might include a tag
            new_pc = AArch64.BranchAddr(new_pc);
    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType_ERET);
    else
        BranchToAddr(new_pc, BranchType_ERET);
```

Library pseudocode for aarch64/instrs/countop/CountOp

```c
enumeration CountOp     {
    CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```
Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend  

```
// DecodeRegExtend()  
// ================  
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)  
    case op of  
        when '000' return ExtendType_UXTB;  
        when '001' return ExtendType_UXTH;  
        when '010' return ExtendType_UXTW;  
        when '011' return ExtendType_UXTX;  
        when '100' return ExtendType_SXTB;  
        when '101' return ExtendType_SXTH;  
        when '110' return ExtendType_SXTW;  
        when '111' return ExtendType_SXTX;
```

Library pseudocode for aarch64/instrs/extendreg/ExtendReg  

```
// ExtendReg()  
// ===========  
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift)  
    assert shift >= 0 && shift <= 4;  
    bits(N) val = X[reg];  
    boolean unsigned;  
    integer len;  
    case exttype of  
        when ExtendType_SXTB unsigned = FALSE; len = 8;  
        when ExtendType_SXTH unsigned = FALSE; len = 16;  
        when ExtendType_SXTW unsigned = FALSE; len = 32;  
        when ExtendType_SXTX unsigned = FALSE; len = 64;  
        when ExtendType_UXTB unsigned = TRUE; len = 8;  
        when ExtendType_UXTH unsigned = TRUE; len = 16;  
        when ExtendType_UXTW unsigned = TRUE; len = 32;  
        when ExtendType_UXTX unsigned = TRUE; len = 64;  
    len = Min(len, N - shift);  
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);  
```

Library pseudocode for aarch64/instrs/extendreg/ExtendType  

```
enumeration ExtendType  
    {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,  
    ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

Library pseudocode for aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp  

```
enumeration FPMaxMinOp  
    {FPMaxMinOp_MAX, FPMaxMinOp_MIN,  
    FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

Library pseudocode for aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp  

```
enumeration FPUnaryOp  
    {FPUnaryOp_ABS, FPUnaryOp_MOV,  
    FPUnaryOp_NEG, FPUnaryOp_SQRT};
```
**Library pseudocode for aarch64/instrs/float/convert/fpconvop/FPConvOp**

```plaintext
enumeration FPConvOp {
    FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
    FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF,
    FPConvOp_CVT_FtoI_JS
};
```

**Library pseudocode for aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred**

```plaintext
// 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;
```

Shared Pseudocode Functions
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);

mask = ((tmask
AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));

mask = ((tmask
AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));

mask = ((tmask
AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));

mask = ((tmask
AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));

mask = ((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

Shared Pseudocode Functions
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));

  return FALSE;
// DecodeShift()
// ============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
  case op of
    when '00' return ShiftType_LSL;
    when '01' return ShiftType_LSR;
    when '10' return ShiftType_ASR;
    when '11' return ShiftType_ROR;

// ShiftReg()
// =========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount)
  bits(N) result = X[reg];
  case shiftype of
  when ShiftType_LSL result = LSL(result, amount);
  when ShiftType_LSR result = LSR(result, amount);
  when ShiftType_ASR result = ASR(result, amount);
  when ShiftType_ROR result = ROR(result, amount);
  return result;

// ShiftType
// ===========

enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};

// LogicalOp
// ===========

enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

// MemAtomicOp
// =============


// MemOp
// ========

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
Library pseudocode for aarch64/instrs/memory/prefetch/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_WFET,
    SystemHintOp_WFIT,
    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_SSSB,
    PSTATEField_TCO, // Armv8.5
    PSTATEField_SP
};
// SysOp()
// ========
SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT // S1E1R
        when '100 0111 1000 000' return Sys_AT // S1E2R
        when '110 0111 1000 000' return Sys_AT // S1E3R
        when '000 0111 1000 001' return Sys_AT // S1E1W
        when '100 0111 1000 001' return Sys_AT // S1E2W
        when '110 0111 1000 001' return Sys_AT // S1E3W
        when '000 0111 1000 010' return Sys_AT // S1E0R
        when '000 0111 1000 011' return Sys_AT // S1E0W
        when '100 0111 1000 100' return Sys_AT // S12E1R
        when '100 0111 1000 101' return Sys_AT // S12E1W
        when '100 0111 1000 110' return Sys_AT // S12E0R
        when '100 0111 1000 111' return Sys_AT // S12E0W
        when '011 0111 0100 001' return Sys_DC // ZVA
        when '000 0111 0110 001' return Sys_DC // IVAC
        when '000 0111 0110 010' return Sys_DC // ISW
        when '011 0111 1010 001' return Sys_DC // CVAC
        when '000 0111 1010 010' return Sys_DC // CSW
        when '011 0111 1010 011' return Sys_DC // CVAU
        when '011 0111 1110 010' return Sys_DC // CIVAC
        when '011 0111 1110 011' return Sys_DC // CISW
        when '011 0111 1101 001' return Sys_DC // CVADP
        when '000 1000 0001 000' return Sys_IC // IALLUIS
        when '000 1000 0101 000' return Sys_IC // IALLU
        when '011 0111 0101 001' return Sys_IC // IVAU
        when '100 1000 0000 001' return Sys_IC // IPAS2E1IS
        when '100 1000 0000 101' return Sys_IC // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_IC // VMALLE1IS
        when '100 1000 0011 000' return Sys_IC // ALLE2IS
        when '110 1000 0011 000' return Sys_IC // ALLE3IS
        when '000 1000 0011 001' return Sys_IC // VAE1IS
        when '100 1000 0011 001' return Sys_IC // VAE2IS
        when '110 1000 0011 001' return Sys_IC // VAE3IS
        when '000 1000 0011 010' return Sys_IC // ASIDE1IS
        when '000 1000 0011 011' return Sys_IC // ASIDE1IS
        when '100 1000 0011 100' return Sys_IC // ASIDE1IS
        when '100 1000 0011 101' return Sys_IC // VAE2IS
        when '110 1000 0011 101' return Sys_IC // VAE3IS
        when '000 1000 0011 110' return Sys_IC // VALLE1IS
        when '000 1000 0011 111' return Sys_IC // VALE2IS
        when '100 1000 0011 111' return Sys_IC // VALE3IS
        when '100 1000 0011 110' return Sys_IC // VMALLS12E1IS
        when '100 1000 0011 111' return Sys_IC // VMALLS12E1IS
    return Sys_SYS;

Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
constant bits(16) ASID_NONE = Zeros();

// Broadcast
// =========
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
domain.

Broadcast(Shareability shareability, TLBIREcord r)
    IMPLEMENTATION_DEFINED;

boolean HasLargeAddress(Regime regime)
    if !Have52BitIPAAndPASpaceExt() then
        return FALSE;
    case regime of
        when Regime_EL3
            return TCR_EL3<32> == '1';
        when Regime_EL2
            return TCR_EL2<32> == '1';
        when Regime_EL20
            return TCR_EL2<59> == '1';
        when Regime_EL10
            return TCR_EL1<59> == '1';
        otherwise
            Unreachable();

enumeration Regime {
    Regime_EL10,           // EL1&0
    Regime_EL20,           // EL2&0
    Regime_EL2,            // EL2
    Regime_EL3             // EL3
};

enumeration SecurityState {
    SS_NonSecure,
    SS_Secure
};
// SecurityStateAtEL()
// ===================
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
    if !HaveEL(EL3) then
        if boolean IMPLEMENTATION_DEFINED "Secure-only implementation" then
            return SS_Secure;
        else
            return SS_NonSecure;
    elsif EL == EL3 then
        return SS_Secure;
    else
        // For EL call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if SCR_EL3.NS == '1' then SS_NonSecure else SS_Secure;
        else
            return if SCR.NS == '1' then SS_NonSecure else SS_Secure;
    end

Library pseudocode for aarch64/instrs/system/sysops/tlbi/Shareability

enumeration Shareability {
    Shareability_None,
    Shareability_Inner,
    Shareability_Outer
};

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI

// TLBI
// =====
// IMPLEMENTATION DEFINED TLBI function.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBILevel

enumeration TLBILevel {
    TLBILevel_Any,
    TLBILevel_Last
};

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIOp

enumeration TLBIOp {
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
};
type TLBIRecord is (  
  TLBIOp op,  
  SecurityState security,  
  Regime regime,  
  bits(16) vmid,  
  bits(16) asid,  
  TLBILevel level,  
  TLBI_MemAttr attr,  
  FullAddress address,  // VA/IPA/BaseAddress  
  bits(64) end_address,  // for range operations, end address  
  bits(2) tg,  // for range - the TG parameter  
  bits(4) ttl,  
)

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_ALL

// TLBI_ALL()  
// =========  
// Invalidates all entries for the indicated translation regime with the  
// the indicated security state for all TLBs within the indicated shareability domain.  
// Invalidation applies to all applicable stage 1 and stage 2 entries.  
// The indicated attr defines the attributes of the memory operations that must be completed in  
// order to deem this operation to be completed.  
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation  
// are required to complete.

TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBI_MemAttr attr)  
assert PSTATE.EL IN (EL3, EL2);

  TLBIRecord r;
  r.op           = TLBIOp_ALL;
  r.security     = security;
  r.regime       = regime;
  r.level        = TLBILevel_Any;
  r.attr         = attr;
  TLBI(r);
  if shareability != Shareability_None then Broadcast(shareability, r);
return;
Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_ASID

// TLBI_ASID()
// ===========
// Invalidates all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
        TLBI_MemAttr attr, bits(64) Xt)

assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op           = TLBIOp_ALL;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    TLBI(r);

if shareability != Shareability_None then Broadcast(shareability, r);
return;

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_IPAS2

// TLBI_IPAS2()
// ============
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
            Shareability shareability, TLBI_MemAttr attr, bits(64) Xt)

assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address.address = Xt<39:0> : Zeros(12);
    r.address.NS      = if security == SS_NonSecure then '1' else Xt<63>;
    TLBI(r);

if shareability != Shareability_None then Broadcast(shareability, r);
return;
**Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_MemAttr**

```c
enumeration TLBI_MemAttr {
    TLBI_AllAttr,
    TLBI_ExcludeXS
}
```

**Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_RIPAS2**

```c
// TLBI_RIPAS2()
// =============
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op           = TLBIOp_RIPAS2;
r.security     = security;
r.regime       = regime;
r.vmid         = vmid;
r.level        = level;
r.attr         = attr;
r.ttl          = Xt<47:44>;
bits(2) tg        = Xt<47:46>;
integer scale     = UInt(Xt<45:44>);
integer num       = UInt(Xt<43:39>);
integer baseaddr  = SInt(Xt<36:0>);

bits(64) start_address;
boolean valid;
(valid, r.tg, start_address, r.end_address) = TLBI_Range(regime, Xt);
if !valid then return;

r.address.address = start_address<51:0>;

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;
```
// TLBI_RVA()
// =========
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
          Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBRecord r;
r.op     = TLBIOp_RVA;
r.security = security;
r.regime   = regime;
r.vmid     = vmid;
r.level    = level;
r.attr     = attr;
r.asid     = Xt<63:48>;
r.ttl      = Xt<47:44>;

bits(64) start_address;
boolean valid;

(valid, r.tg, start_address, r.end_address) = TLBI_Range(regime, Xt);
if !valid then return;

r.address.address = start_address<51:0>;
r.address.NS      = if security == SS_NonSecure then '1' else Xt<63>;

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;
// TLBI_RVAA()
// ===========
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};
TLBIRecord r;
r.op = TLBIOp_RVAA;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.ttl = Xt<47:44>;

bits(2) tg = Xt<47:46>;
integer scale = UInt(Xt<45:44>);
integer num = UInt(Xt<43:39>);
integer baseaddr = SInt(Xt<36:0>);

bits(64) start_address;
boolean valid;

(valid, r.tg, start_address, r.end_address) = TLBI_Range(regime, Xt);
if !valid then return;

r.address.address = start_address<51:0>;
TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;
// TLBI_Range()
// ============
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBI_Range(Regime regime, bits(64) Xt)

    boolean valid = TRUE;
    bits(64) start = Zeros(64);
    bits(64) end = Zeros(64);

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start, end);
    end if

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            else
                start<48:12> = Xt<36:0>;
                start<63:49> = Replicate(Xt<36>, 15);
            end if
        end when

        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            else
                start<50:14> = Xt<36:0>;
                start<63:51> = Replicate(Xt<36>, 13);
            end if
        end when

        when '11' // 64KB
            tg_bits = 16;
            start<52:16> = Xt<36:0>;
            start<63:53> = Replicate(Xt<36>, 11);
        end when

        otherwise
            Unreachable();
        end case

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end = start + range<63:0>;

    if end<52> != start<52> then
        // overflow, saturate it
        end = Replicate(start<52>, 64-52) : Ones(52);
    end if

    return (valid, tg, start, end);
// TLBI VA()
// =========
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI VA(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBI0p VA;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.asid = Xt<63:48>;
r.ttl = Xt<47:44>;

r.address.address = Xt<39:0> : Zeros(12);

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;
// TLBI_VAA()
// ===========
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//   TLBILevel_Any : this applies to TLB entries at all levels
//   TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)

assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op = TLBIOp_VAA;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl = Xt<47:44>;
    r.address.address = Xt<39:0> : Zeros(12);

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;

// TLBI_VMALL()
// ============
// Invalidates all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBI_MemAttr attr)

assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op = TLBIOp_VMALL;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel_Any;
    r.vmid = vmid;
    r.attr = attr;

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;
Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_VMALLS12

```c
// TLBI_VMALLS12()
// ===============
// Invalidates all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBI_MemAttr attr)
assert PSTATE.EL IN {EL3, EL2};
TLBIRecord r;
    r.op           = TLBIOp_VMALLS12;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel.Any;
    r.vmid         = vmid;
    r.attr         = attr;
    TLBI(r);
if shareability != Shareability.None then Broadcast(shareability, r);
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID

```c
// VMID[]
// ======
// Effective VMID.

bits(16) VMID[]
if EL2Enabled() then
    return VTTBR_EL2.VMID;
elsif HaveEL(EL2) && HaveSecureEL2Ext() then
    return Zeros(16);
else
    return VMID_NONE;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID_NONE

```c
constant bits(16) VMID_NONE = Zeros();
```

Library pseudocode for aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```c
enumeration VBitOp {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

Library pseudocode for aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```c
enumeration CompareOp {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
    CompareOp_LE, CompareOp_LT};
```

Library pseudocode for aarch64/instrs/vector/logical/immediateop/ImmediateOp

```c
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
```
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
  boolean altfp = HaveAltFP() && !UsingAArch32() && FPCR.AH == '1';
  return Reduce(op, input, esize, altfp);

// Reduce()
// ========
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behaviour.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
  integer half;
  bits(esize) hi;
  bits(esize) lo;
  bits(esize) result;

  if N == esize then
    return input<esize-1:0>;
  half = N DIV 2;
  hi = Reduce(op, input<N-1:half>, esize, altfp);
  lo = Reduce(op, input<half-1:0>, esize, altfp);

  case op of
    when ReduceOp_FMINNUM
      result = FPMinNum(lo, hi, FPCR[]);
    when ReduceOp_FMAXNUM
      result = FPMaxNum(lo, hi, FPCR[]);
    when ReduceOp_FMIN
      result = FPMin(lo, hi, FPCR[], altfp);
    when ReduceOp_FMAX
      result = FPMax(lo, hi, FPCR[], altfp);
    when ReduceOp_FADD
      result = FPAdd(lo, hi, FPCR[]);
    when ReduceOp_ADD
      result = lo + hi;

  return result;

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
  ReduceOp_FMIN, ReduceOp_FMAX,
  ReduceOp_FADD, ReduceOp_ADD};
Library pseudocode for aarch64/translation/attrs/AArch64.CombineS1S2Desc

// AArch64.CombineS1S2Desc()
// =========================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor AArch64.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc, AccType s2acctype)

AddressDescriptor result;
result.paddress = s2desc.paddress;

apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
if IsFault(s1desc) || IsFault(s2desc) then
  result = if IsFault(s1desc) then s1desc else s2desc;
else
  result.fault = AArch64.NoFault();
  if s2desc.memattrs.memtype == MemType_Device ||
     (!apply_force_writeback && s1desc.memattrs.memtype == MemType_Device) ) then
    result.memattrs.memtype = MemType_Device;
  if s1desc.memattrs.memtype == MemType_Normal then
    result.memattrs.device = s2desc.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);
  result.memattrs.tagged = FALSE;
  // S1 can be either Normal or Device, S2 is Normal.
else
  result.memattrs.memtype = MemType_Normal;
  result.memattrs.device = DeviceType_UNKNOWN;
result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner, s2acctype);
result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer, s2acctype);
result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
result.memattrs.outershareable = (s1desc.memattrs.outershareable || s2desc.memattrs.outershareable);
result.memattrs.tagged = (s1desc.memattrs.tagged &&
                           result.memattrs.inner.attrs == MemAttr_WB &&
                           result.memattrs.inner.hints == MemHint_RWA &&
                           result.memattrs.outer.attrs == MemAttr_WB &&
                           result.memattrs.outer.hints == MemHint_RWA);
result.memattrs = MemAttrDefaults(result.memattrs);

return result;
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_INTRDEVICE);
    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);
    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 WB-RWA
            memattrs.memtype = MemType_Normal;
            memattrs.outer = LongConvertAttrsHints('1111', accType);  // WB_RWA
            memattrs.inner = LongConvertAttrsHints('1111', accType);  // WB_RWA
            memattrs.shareable = SH<1> == '1';
            memattrs.outershareable = SH == '10';
            memattrs.tagged = TRUE;
        else
            Unreachable();                          // Reserved, handled above
        end
    end
    if ((HCR_EL2.VM == '1' || HCR_EL2.DC == '1') &&
        (PSTATE.EL == EL1 || (PSTATE.EL == EL0 && HCR_EL2.TGE == '0')) &&
        accType != AccType_NV2REGISTER ) then
        return memattrs;
    else
        return MemAttrDefaults(memattrs);
    end

Shared Pseudocode Functions
Library 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 actype, boolean iswrite)
assert !ELUsingAArch32(S1TranslationRegime());

TLBRecord result;
result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;

Top = AddrTop(vaddress, (actype == AccType_IFETCH), PSTATE.EL);
if !IsZero(vaddress<Top:PAMax>()) then
  level = 0;
  ipaddress = bits(52) UNKNOWN;
  secondstage = FALSE;
  s2fs1walk = FALSE;
  result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,boolean UNKNOWN, level, actype, 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.ininner.attrs = MemAttr_WB;  // Write-back
  result.addrdesc.memattrs.inner.hints = MemHint_RWA;
  result.addrdesc.memattrs.shareable = FALSE;
  result.addrdesc.memattrsoutershareable = FALSE;
  result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
elsif actype != 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
  cacheable = SCTLR().I == '1';
  result.addrdesc.memattrs.memtype = MemType_Normal;
  if cacheable then
    result.addrdesc.memattrs.ininner.attrs = MemAttr_WT;
    result.addrdesc.memattrs.inner.hints = MemHint_RA;
  else
    result.addrdesc.memattrs.ininner.attrs = MemAttr_NC;
    result.addrdesc.memattrs.inner.hints = MemHint_No;
  result.addrdesc.memattrs.shareable = TRUE;
  result.addrdesc.memattrsoutershareable = TRUE;
  result.addrdesc.memattrs.tagged = FALSE;
result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;
result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);
result.perms.ap = bits(3) UNKNOWN;
result.perms.xn = '0';
result.perms.pxn = '0';
result.nG = bit UNKNOWN;
result.contiguous = boolean UNKNOWN;
result.domain = bits(4) UNKNOWN;
result.level = integer UNKNOWN;
result.blocksize = integer UNKNOWN;
result.addrdesc.paddress.address = vaddress<51:0>;
result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
result.addrdesc.fault = AArch64.NoFault();
result.descupdate.descaddr = result.addrdesc;
return result;

**Library pseudocode for aarch64/translation/checks/AArch64.AccessUsesEL**

```pseudocode
// 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;
```

```
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 !ELUsingAAarch32(S1TranslationRegime());

wxn = SCTLR[].WXN == '1';

if (PSTATE.EL == EL0 || IsInHost() || (PSTATE.EL == EL1 && !HaveNV2Ext()) || (PSTATE.EL == EL1 && HaveNV2Ext() && (acctype != AccType_NV2REGISTER || !ELIsInHost(EL2)))) then

priv_r = TRUE;
priv_w = perms.ap<2> == '0';
user_r = perms.ap<1> == '1';
user_w = perms.ap<2:1> == '01';

ispriv = AArch64.AccessUsesEL(acctype) != EL0;

user_xn = perms.xn == '1' || (user_w && wxn);
priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;
pan = if HavePANExt() then PSTATE.PAN else '0';
epan = if HavePAN3Ext() then SCTLR[].EPAN else '0';

// Restriction on Secure instruction fetch
if boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects PAN3 execute permission check" then
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        user_xn = TRUE;
        priv_xn = TRUE;

if (EL2Enabled()) && ((PSTATE.EL == EL1 && HaveNVExt()) && HCR_EL2.<NV, NV1> == '11') || (HaveNV2Ext() && acctype == AccType_NV2REGISTER && HCR_EL2.NV2 == '1')) then

    pan = '0';

is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
is_ats1xp = (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr());

if (pan == '1'. && (user_r || (epan == '1' && !user_xn)) &&
    ispriv && (is_ldst || is_ats1xp)) then

    priv_r = FALSE;
    priv_w = FALSE;

    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 !boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects PAN3 execute permission check" then
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        xn = TRUE;

if acctype == AccType_IFETCH then
    fail = xn;
    failedread = TRUE;
elsif acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW} then
    fail = !r || !w;
    failedread = !r;
elsif iswrite then
    fail = !w;
    failedread = FALSE;
elsif acctype == AccType_DC && PSTATE.EL != EL0 then
    // DC maintenance instructions operating by VA, cannot fault from stage 1 translation,
    // other than DC IVAC, which requires write permission, and operations executed at EL0,
// which require read permission.
fail = FALSE;
else
  fail = !r;
  failedread = TRUE;

if fail then
  secondstage = FALSE;
s2fs1walk = FALSE;
  ipaddress = bits(52) UNKNOWN;
return AArch64.PermissionFault(ipaddress,boolean UNKNOWN, level, acctype, !failedread, secondstage, s2fs1walk);
else
  return AArch64.NoFault();

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;
  elseiswrite && !s2fs1walk then
    fail = !w;
    failedread = !iswrite;
  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)
    FaultRecord fault = AArch64.NoFault();
    d_side = (acctype != AccType_IFETCH);
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        mask = '0';
        generate_exception = AArch64.GenerateDebugExceptionsFrom(EL2, IsSecure(), mask) && MDSCR_EL1.MDE == '1';
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
        halt = HaltOnBreakpointOrWatchpoint();
    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, acctype, size);
    return fault;
Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
if acctype IN {AccType_TTW, AccType_IC, AccType_AT} then
    return AArch64.NoFault();
if acctype == AccType_DC then
    if !iswrite then
        return AArch64.NoFault();
match = FALSE;
ispriv = AArch64.AccessUsesEL(acctype) != EL0;
for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
    match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite);
if match && HaltOnBreakpointOrWatchpoint() then
    if acctype != AccType_NONFAULT && acctype != AccType_CNOTFIRST then
        reason = DebugHalt_Watchpoint;
        EDWAR = vaddress;
        Halt(reason);
    else
        // Fault will be reported and cancelled
        return AArch64.DebugFault(acctype, iswrite);
else
    return AArch64.NoFault();

Library pseudocode for aarch64/translation/faults/AArch64.AccessFlagFault

// AArch64.AccessFlagFault()
// =========================

FaultRecord AArch64.AccessFlagFault(bits(52) ipaddress,boolean NS, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)
extflag = bit UNKNOWN;
errortype = bits(2) UNKNOWN;
return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress, NS, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.AddressSizeFault

// AArch64.AddressSizeFault()
// =========================

FaultRecord AArch64.AddressSizeFault(bits(52) ipaddress,boolean NS, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)
extflag = bit UNKNOWN;
errortype = bits(2) UNKNOWN;
return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress, NS, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
Library pseudocode for aarch64/translation/faults/AArch64.AlignmentFault

// AArch64.AlignmentFault()
// ========================
FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.AsynchExternalAbort

// AArch64.AsynchExternalAbort()
// =============================
// Wrapper function for asynchronous external aborts
FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)
    faulttype = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
    return AArch64.CreateFaultRecord(faulttype, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.DebugFault

// AArch64.DebugFault()
// ====================
FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)
    ipaddress = bits(52) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.ExclusiveFault

// AArch64.ExclusiveFault()
// ========================
FaultRecord AArch64.ExclusiveFault(AccType acctype, boolean iswrite)
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = boolean UNKNOWN;
    s2fs1walk = boolean UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Exclusive, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
// AArch64.NoFault()
// ===============
FaultRecord AArch64.NoFault()
  ipaddress = bits(52) UNKNOWN;
  level = integer UNKNOWN;
  acctype = AccType_NORMAL;
  iswrite = boolean UNKNOWN;
  extflag = bit UNKNOWN;
  errortype = bits(2) UNKNOWN;
  secondstage = FALSE;
  s2fs1walk = FALSE;
  return AArch64.CreateFaultRecord(Fault_None, ipaddress, boolean UNKNOWN, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);

// AArch64.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);

// 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 || ConstranUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE then
    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);
desc = _Mem[descaddr2, 8, accdesc, iswrite];
case el of
  when EL3
    reversedescriptors = SCTLR_EL3.EE == '1';
  when EL2
    reversedescriptors = SCTLR_EL2.EE == '1';
  otherwise
    reversedescriptors = SCTLR_EL1.EE == '1';
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';
  fi

  TLBRecord S1;
  S1.addrdesc.fault = AArch64.NoFault();
  ipaddress = bits(52) UNKNOWN;
  secondstage = FALSE;
  s2fs1walk = FALSE;

  if s1_enabled then                         // First stage enabled
    S1 = AArch64.TranslationTableWalk(ipaddress, TRUE, vaddress, acctype, iswrite, secondstage,
      s2fs1walk, size);
    permissioncheck = TRUE;
    if acctype == AccType_IFETCH then
      InGuardedPage = S1.GP == '1';      // Global state updated on instruction fetch that denotes
    // if the fetched instruction is from a guarded page.
    else
      S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
      permissioncheck = FALSE;
      InGuardedPage = FALSE;                 // No memory is guarded when stage 1 address translation is
  fi

  if !IsFault(S1.addrdesc) && UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr then
    if S1.addrdesc.memattrs.memtype == MemType_Device && S1.addrdesc.memattrs.device != DeviceType_GRE 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_IFETCH) || acctype == AccType_DCZVA) &&
          !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device) then
          S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
        fi
      fi
    fi

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device &&
      acctype == AccType_IFETCH) then
      S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
        acctype, iswrite, secondstage, s2fs1walk);
    fi

    // 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);
  fi

  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;
        hwupdatet combined  = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                               size, hwupdatetwalk);
    else
        result = S1;
    return result;
Library pseudocode for aarch64/translation/translation/AArch64.SecondStageTranslate

// 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 s2fslwalk, 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,
                                         s2fslwalk, size);

    // Check for unaligned data accesses to Device memory
    if (((!wasaligned && acctype != AccType_IFETCH) ||
         (acctype == AccType_DCZVA && !s2fslwalk)) &&
        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,s2fslwalk, hwupdatewalk);

    // Check for instruction fetches from Device memory not marked as execute-never. As there
    // has not been a Permission Fault then the memory is not marked execute-never.
    if (!s2fslwalk && !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, s2fslwalk);

    if (s2fslwalk && !IsFault(S2.addrdesc) &&
        S2.addrdesc.memattrs.memtype == MemType_Device) then
        // Check for protected table walk.
        if HCR_EL2.PTW == '1' then
            S2.addrdesc.fault = AArch64.PermissionFault(ipaddress,
                                                        NS, S2.level,
                                                        acctype, iswrite, secondstage, s2fslwalk);
        else
            // Translation table walk occurs as Normal Non-cacheable memory.
            S2.addrdesc.memattrs.memtype = MemType_Normal;
            S2.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            S2.addrdesc.memattrs.outer.attrs = MemAttr_NC;
            S2.addrdesc.memattrs.shareable = TRUE;
            S2.addrdesc.memattrs.outershareable = TRUE;

        // Check and update translation table descriptor if required
        S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault,
                                                            secondstage, vaddress, acctype,
                                                            iswrite, s2fslwalk, hwupdatewalk);

    if s2fslwalk then
        result = AArch64.CombineS1S2Desc(S1, S2.addrdesc, AccType_TTW);
    else
        result = AArch64.CombineS1S2Desc(S1, S2.addrdesc, acctype);
    else
        result = S1;

return result;
Library pseudocode for aarch64/translation/translation/AArch64.SecondStageWalk

// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype, boolean iswrite, integer size, boolean hwupdatewalk)

assert HasS2Translation();

s2fs1walk = TRUE;
wasaligned = TRUE;
return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk, size, hwupdatewalk);

Library pseudocode for aarch64/translation/translation/AArch64.TranslateAddress

// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
if !IsFault(result) then
    result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(vaddress);

return result;
Library pseudocode for aarch64/translation/walk/AArch64.TranslationTableWalk
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 accctype, boolean iswrite, boolean secondstage,
   boolean s2fslwalk, integer size)
if !secondstage then
   assert !ELUsingAArch32(S1TranslationRegime());
else
   assert (IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2)) && HasS2Translation());

TLBRecord result;
AddressDescriptor descaddr;
bits(64) baseregister;
bits(64) inputaddr; // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
bit nswalk; // Stage 2 translation table walks are to Secure or to Non-secure PA space

result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;
descaddr.memattrs.memtype = MemType_Normal;

// Derived parameters for the translation table walk:
// grainsize = Log2(Size of Table) - Size of Table is 4KB, 16KB or 64KB in AArch64
// stride = Log2(Address per Level) - Bits of address consumed at each level
// firstblocklevel = First level where a block entry is allowed
// ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
// inputsize = Log2(Size of Input Address) - Input Address size in bits
// level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
   // First stage translation
   inputaddr = ZeroExtend(vaddress);
   el = AArch64.AccessUsesEL(accctype);
   isprivileged = AArch64.AccessUsesEL(accctype) != EL0;
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;
      if !Have52BitVAExt() && inputsize > inputsize_max then
         c = ConstrinUnpredictable(Unpredictable_RESTnSZ);
         assert c IN {Constraint_FORCE, Constraint_FAULT};
         if c == Constraint_FORCE then inputsize = inputsize_max;
      if inputsize < inputsize_min then
         c = ConstrinUnpredictable(Unpredictable_RESTnSZ);
         assert c IN {Constraint_FORCE, Constraint_FAULT};
         if c == Constraint_FORCE then inputsize = inputsize_min;
      ps = TCR_EL3.PS;
      basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top>);
      disabled = false;
      baseregister = TTBR0_EL3;
      descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
      reversedescriptors = SCTLR_EL3.EE == '1';
      lookupsecure = true;
      singlepriv = false;
      update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.HA == '1';
      update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
      hierattrdisenabled = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
   elsif ELIsInHost(el) then
      if inputaddr<top> == '0' then
         // Shared Pseudocode Functions
      end
   end
largegrain = TCR_EL2.TG0 == '01';
midgrain = TCR_EL2.TG0 == '10';
inputsize_max = 64 - UInt(TCR_EL2.T0SZ);
inputsize_max = if Have52BitVAExt() & largegrain then 52 else 48;

if !Have52BitVAExt() & inputsize > inputsize_max then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_max;

inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_min;

basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top:inputsize>);

disabled = TCR_EL2.EPD0 == '1' || (!isprivileged & HaveE0PDExt() && TCR_EL2.E0PD0 == '1');
if el == EL0 & TCR_EL2.NFD0 == '1' then
    disabled = disabled || acctype == AccType_NONFAULT;
    baseregister = TTBR0_EL2;
    descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
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 !HaveSmallTranslationTableExt() & largegrain then 52 else 48;

if !Have52BitVAExt() & inputsize > inputsize_max then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_max;

inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_min;

basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputaddr<top:inputsize>);

disabled = TCR_EL2.EPD1 == '1' || (!isprivileged & HaveE0PDExt() && TCR_EL2.E0PD1 == '1');
if el == EL0 & TCR_EL2.NFD1 == '1' then
    disabled = disabled || acctype == AccType_NONFAULT;
    baseregister = TTBR1_EL2;
    descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH1, TCR_EL2.ORGN1, TCR_EL2.IRGN1, secondstage);
else
    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;

if !Have52BitVAExt() & inputsize > inputsize_max then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_max;

inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_min;

ps = TCR_EL2.IPS;
reversedescriptors = SCTLR_EL2.EE == '1';
lookupsecure = if IsSecureEL2Enabled() then IsSecure() else FALSE;
singlepriv = FALSE;
update_AF = HaveAccessFlagUpdateExt() & TCR_EL2.HA == '1';
update_AP = HaveDirtyBitModifierExt() & update_AF & TCR_EL2.HD == '1';
else if 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;

if !Have52BitVAExt() & inputsize > inputsize_max then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_max;

inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_min;

ps = TCR_EL2.PS;
baseregister = TTBR0_EL2;
descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
reversedescriptors = SCTLR_EL2.EE == '1';
lookupsecure = if IsSecureEL2Enabled() then IsSecure() else FALSE;
singlepriv = TRUE;
update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD == '1';
else
if inputaddr<top> == '0' then
  inputsize = 64 - UInt(TCR_EL1.T0SZ);
largegrain = TCR_EL1.TG0 == '01';
middigrain = TCR_EL1.TG0 == '10';
inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
if !Have52BitVAExt() && inputsize > inputsize_max then
  c = Construnpredictable(Unpredictable_RESTnSZ);
  assert c IN {Constraint_FORCE, Constraint_FAULT};
  if c == Constraint_FORCE then inputsize = inputsize_max;

inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 52 else 48)
if inputsize < inputsize_min then
  if largegrain then
    grainsize = 16;                                             // Log2(64KB page size)
    firstblocklevel = (if Have52BitPAExt() then 1 else 2);      // Largest block is 4TB (2^42 bytes)
  elsif middigrain then
    grainsize = 14;                                             // Log2(16KB page size)
    firstblocklevel = 2;                                        // Largest block is 32MB (2^25 bytes)
  else // Small grain
    grainsize = 12;                                             // Log2(4KB page size)
    firstblocklevel = 1;                                        // Largest block is 1GB (2^30 bytes)
  stride = grainsize - 3;                                         // Log2(page size / 8 bytes)
// The starting level is the number of strides needed to consume the input address
// level = 4 - (1 + ((inputsize - grainsize - 1) DIV stride));

else
  // Second stage translation
  inputaddr = ZeroExtend(ipaddress);
  if IsSecureBelowEL3() then
    // Second stage for Secure translation regime
    if s1_nonsecure then // Non-secure IPA space
      t0size = VTCR_EL2.T0SZ;
      tg0 = VTCR_EL2.TG0;
      nswalk = VTCR_EL2.NSW;
    else // Secure IPA space
      t0size = VSTCR_EL2.T0SZ;
      tg0 = VSTCR_EL2.TG0;
      nswalk = VSTCR_EL2.SW;
    // Stage 2 translation accesses the Non-secure PA space or the Secure PA space
    if nswalk == '1' then
      // When walk is Non-secure, access must be to the Non-secure PA space
      nsaccess = '1';
    elsif !s1_nonsecure then
      // When walk is Secure and in the Secure IPA space,
      // access is specified by VSTCR_EL2.SA
      nsaccess = VSTCR_EL2.SA;
    elsif VSTCR_EL2.SW == '1' || VSTCR_EL2.SA == '1' then
      // When walk is Secure and in the Non-secure IPA space,
      // access is Non-secure when VSTCR_EL2.SA specifies the Non-secure PA space
      nsaccess = '1';
    else
      // When walk is Secure and in the Non-secure IPA space,
      // if VSTCR_EL2.SA specifies the Secure PA space, access is specified by VTCR_EL2.NSA
      nsaccess = VTCR_EL2.NSA;
  else
    // Second stage for Non-secure translation regime
    t0size = VTCR_EL2.T0SZ;
    tg0 = VTCR_EL2.TG0;
    nswalk = '1';
    nsaccess = '1';
  inputsize = 64 - UInt(t0size);
  largegrain = tg0 == '01';
  midgrain = tg0 == '10';

  inputsize_max = if Have52BitPAExt() && PAMax() == 52 && largegrain then 52 else 48;
  if !Have52BitPAExt() && inputsize > inputsize_max then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_max;

  inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 47 else 48);
  if inputsize < inputsize_min then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_min;

  ps = VTCR_EL2.PS;
  basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<63:inputsize>);
  descmemattrs =WalkAttrDecode(VTCR_EL2.SH0, VTCR_EL2.ORGN0, VTCR_EL2.IRGN0, secondstage);
  reversedescriptors = SCTLR_EL2.EE == '1';
  singlepriv = TRUE;
  update_AF = HaveAccessFlagUpdateExt() && VTCR_EL2.HA == '1';
  update_AP = HaveDirtyBitModifierExt() && update_AF && VTCR_EL2.HD == '1';

  if IsSecureEL2Enabled() then
    lookupsecure = !s1_nonsecure;
  else
    lookupsecure = FALSE;

  if lookupsecure then

baseregister = VSTTBR_EL2;
    startlevel   = UInt(VSTCR_EL2 SL0);
else
    baseregister = VTTBR_EL2;
    startlevel   = UInt(VTCR_EL2 SL0);
if largegrain then
    grainsize = 16;                                         // Log2(64KB page size)
    level = 3 - startlevel;
    firstblocklevel = (if Have52BitPAExt() then 1 else 2);  // Largest block is 4TB (2^42 bytes)
    // and 512MB (2^29 bytes) otherwise
else if midgrain then
    grainsize = 14;                                         // Log2(16KB page size)
    level = 3 - startlevel;
    firstblocklevel = 2;                                    // Largest block is 32MB (2^25 bytes)
else // Small grain
    grainsize = 12;                                         // Log2(4KB page size)
    if HaveSmallTranslationTableExt() & startlevel == 3 then
        level = startlevel;
        firstblocklevel = 1;                                    // Largest block is 1GB (2^30 bytes)
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
    isizecheck = 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();
                isizecheck = PAMax();
            when Constraint_FORCENOSLCHECK
                // As FORCE, except use the configured inputsize in the size checks below
                inputsize = PAMax();
            when Constraint_FAULT
                // Generate a translation fault
                basefound = FALSE;
            otherwise
                Unreachable();
        case ps of
            when '000' outputsize = 32;
            when '001' outputsize = 36;
            when '010' outputsize = 40;
    return result;
when '011'  outputsize = 42;
when '100'  outputsize = 44;
when '101'  outputsize = 48;
when '110'  outputsize = (if Have52BitPAExt() && largegrain then 52 else 48);
otherwise  outputsize = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

if outputsize > PAMax() then outputsize = PAMax();

if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
  level = 0;
  result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
  return result;

// Bottom bound of the Base address is:
// Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
// (Size of Input Address)/(Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
if outputsize == 52 then
  z = (if baselowerbound < 6 then 6 else baselowerbound);
  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' && SlTranslationRegime();

repeat
  addrselectbottom = (3-level)*stride + grainsize;
  bits(52) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
  descaddr.paddress.address = baseaddress OR index;
  descaddr.paddress.NS = if secondstage then nswalk else ns_table;

  // If there are two stages of translation, then the first stage table walk addresses
  // are themselves subject to translation
  if secondstage || !HasS2Translation() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) then
    descaddr2 = descaddr;
  else
    hwupdatewalk = FALSE;
    descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
    // Check for a fault on the stage 2 walk
    if IsFault(descaddr2) then
      result.addrdesc.fault = descaddr2.fault;
      return result;
  
  // Update virtual address for abort functions
  descaddr2.vaddress = ZeroExtend(vaddress);
  accdesc = CreateAccessDescriptorTTW(acctype, secondstage, s2fs1walk, level);
  desc = _Mem[descaddr2, 8, accdesc, iswrite];
  if reversedescriptors then desc = BigEndianReverse(desc);
  if desc<0> == '0' || (desc<1:0> == '01' && (level == 3 || (HaveBlockBBM() && IsBlockDescriptorNTBitValid() &&
    // 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 && (PAMax() == 52 || boolean IMPLEMENTATION_DEFINED "Address Size Fault on LPA descriptor bits [15:12]")) && !IsZero(desc<15:12>) || (outputsize < 48 && !IsZero(desc<47:outputsize>)) then
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;

    if outputsize == 52 then
        baseaddress = desc<15:12>:desc<47:grainsize>:Zeros(grainsize);
    else
        baseaddress = ZeroExtend(desc<47:grainsize>:Zeros(grainsize));

    if !secondstage then
        // Unpack the upper and lower table attributes
        ns_table    = ns_table    OR desc<63>;

    if !secondstage && !hierattrsdisabled then
        ap_table<1> = ap_table<1> OR desc<62>; // read-only

    if apply_nvnnv1_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_nvnnv1_effect then
            pxn_table   = pxn_table   OR desc<59>;
            ap_table<0> = ap_table<0> OR desc<61>; // privileged

        level = level + 1;
        addrselecttop = addrselectbottom - 1;
        blocktranslate = FALSE;

        until blocktranslate;

    // Check block size is supported at this level
    if level < firstblocklevel then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;

    // Check for misprogramming of the contiguous bit
    if largegrain then
        num_ch_entries = 5;
    elsif midgrain then
        num_ch_entries = if level == 3 then 7 else 5;
    else
        num_ch_entries = 4;

    contiguousbitcheck = inputsize < (addrselectbottom + num_ch_entries);

    if contiguousbitcheck && desc<52> == '1' then
        if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
        return result;

    // Unpack the descriptor into address and upper and lower block attributes
    if largegrain then
        outputaddress = desc<15:12>:desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
    else
        outputaddress = ZeroExtend(desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>);

    // When 52-bit PA is supported, for 64 Kbyte translation granule,
    // block size might be larger than the supported output address size
    if ((outputsize < 52 && !IsZero(outputaddress<51:48>)) && largegrain && (PAMax() == 52 || boolean IMPLEMENTATION_DEFINED "Address Size Fault on LPA descriptor bits [15:12]")) || (outputsize < 48 && !IsZero(outputaddress<47:outputsize>)) then
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;
// Check Access Flag
if desc<10> == '0' then
  if !update_AF then
    result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, sl nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;
  else
    result.descupdate.AF = TRUE;
  end

if update AP && desc<51> == '1' then
  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
end

// Required descriptor if AF or AP[2]/S2AP[2] needs update
result.descupdate.descaddr = descaddr;

if apply_nvnvl_effect then
  pnx = desc<54>; // Bit[54] of the block/page descriptor holds PXN
  xn = '0'; // XN is '0'
  ap = desc<7>:='01';
else
  xn = desc<54>; // Bit[54] of the block/page descriptor holds UXN
  pnx = desc<53>; // Bit[53] of the block/page descriptor holds PXN
  ap = desc<7:6>:='1';
end

contiguousbit = desc<52>:

nG = desc<11>:

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
  // PXN, nG and AP[1] apply in EL1&0 or EL2&0 stage 1 translation regimes if isinglepriv then
  result.perms.ap<1> = ap<1> AND NOT(ap_table<0>); // Force privileged only
  result.perms.pxn = pnx OR pnx_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.pxn = '0';
  result.nG = '0';
result.GP = desc<50>:
  // Stage 1 block or pages might be guarded
  result.perms.ap<0> = '1';
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.xxxn = desc<53>;
  result.perms.pxn = '0';
  result.nG = '0';
  if s2fs1walk then
    result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_TTW);
  else
    result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
end
result.addrdesc.padd.ress NS = nsaccess;
result.addrdesc.address = outputaddress;
result.addrdesc.fault = AArch64.NoFault();
result.contiguous = contiguousbit == '1';
if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;
return result;

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

// ClearStickyErrors()
// ================

ClearStickyErrors()
EDSCR.TXU = '0';            // Clear TX underrun flag
EDSCR.RXO = '0';            // Clear RX overrun flag

if Halted() then // in Debug state
    EDSR.IEO = '0';        // Clear ITR overrun flag

// If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSR.ERR is cleared.
// The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
// in the pseudocode.
if Halted() && EDSR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
    return;
EDSCR.ERR = '0';            // Clear cumulative error flag
return;

Library pseudocode for shared/debug/DebugTarget/DebugTarget

// DebugTarget()
// =============

// Returns the debug exception target Exception level

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

// DebugTargetFrom()
// ================

bits(2) DebugTargetFrom(boolean secure)
    if HaveEL(EL2) && (!secure || (HaveSecureEL2Ext() && HaveEL(EL3) || SCR_EL3.EEL2 == '1')) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elseif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
if !HaveDoubleLock() then
    return FALSE;
elsif ELUsingAArch32(EL1) then
    return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
else
    return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();

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';
        end
    else
        return !IsSecure();
    end
else
    return FALSE;
Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```java
// 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;
        elseif HaveEL(EL3) then
            if ELUsingAArch32(EL3) then
                return SDCR.EPMAD == '0';
            else
                return MDCR_EL3.EPMAD == '0';
            end
        else
            return !IsSecure();
        end
    else
        return FALSE;
}
```

Library pseudocode for shared/debug/authentication/Debug_authentication

```java
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

```java
// 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

```java
// ExternalNoninvasiveDebugAllowed()
// --------------------------------
// Returns TRUE if Trace and PC Sample-based Profiling are allowed
boolean ExternalNoninvasiveDebugAllowed()
{
    return (IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||
            (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1'));
```
Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```java
boolean ExternalNoninvasiveDebugEnabled()
return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```java
boolean ExternalSecureInvasiveDebugEnabled()
if !HaveEL(EL3) && !IsSecure() then return FALSE;
return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```java
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

```java
boolean IsAccessSecure();
```

Library pseudocode for shared/debug/authentication/IsCorePowered

```java
boolean IsCorePowered();
```

// Returns TRUE when an access is Secure
// boolean IsAccessSecure();

// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
// boolean IsCorePowered();
Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained Unpredictable behaviour, otherwise
// returns Constraint_NONE.

(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean isbreakpnt)
    boolean reserved = FALSE;
    // Match 'Usr/Sys/Svc' only valid for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
        reserved = TRUE;
    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
        reserved = TRUE;
    // EL3 is not implemented
    if !HaveEL(EL3) && SSC IN {'01','10'} && HMC:SSC:PxC != '10100' then
        reserved = TRUE;
    // 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','11010'} 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);
CrossTriggerOut enumeration

CrossTriggerOut_DebugRequest,
CrossTriggerOut_IRQ,
CrossTriggerOut_TraceExtIn0,
CrossTriggerOut_TraceExtIn1,
CrossTriggerOut_TraceExtIn2,
CrossTriggerOut_TraceExtIn3;

CrossTriggerIn enumeration

CrossTriggerIn_CrossHalt,
CrossTriggerIn_PMUOverflow,
CrossTriggerIn_RSVD2,
CrossTriggerIn_RSVD3,
CrossTriggerIn_TraceExtOut0,
CrossTriggerIn_TraceExtOut1,
CrossTriggerIn_TraceExtOut2,
CrossTriggerIn_TraceExtOut3;

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()

commrx = (EDSCR.RXfull == '1');
commtx = (EDSCR.TXfull == '0');

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;
Library pseudocode for shared/debug/dccanditr(DBGDTRRX_EL0

// 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 "generate error response";
    return;

if EDSCR.ERR == '1' then return; // Error flag set: ignore write

// The Software lock is OPTIONAL.
if memory_mapped & EDLSR.SLK == '1' then return; // Software lock locked: ignore write

if 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 Halted() & EDSCR.MA == '1' then
    EDSCR.ITE = '0'; // See comments in EDITR[] (external write)
    if !UsingAArch32() then
        ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
        X[1] = bits(64) UNKNOWN;
    else
        ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
        ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "STR R1,[R0],#4"
        R[1] = bits(32) UNKNOWN;
    // 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(64) UNKNOWN;
    else
        "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
        assert EDSCR.RXfull == '0';
        EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
        return;

// 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
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;

underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
value = if underrun then bits(32) UNKNOWN else DTRTX;

if EDSCR.ERR == '1' then return value;
// Error flag set: no side-effects

if memory_mapped && EDLSR.SLK == '1' then
    return value;
// Software lock locked: no side-effects

if underrun then
    EDSCR.TXU = '1';  EDSCR.ERR = '1';
    return value;  // Underrun condition: block side-effects
    // 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 W1,[X0],#4"
else
    ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);
    // T32 "LDR R1,[R0],#4"
// If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
if EDSCR.ERR == '1' then
    EDSCR.TXfull = bit UNKNOWN;
    DBGDTRTX_EL0 = bits(64) UNKNOWN;
else
    if !UsingAArch32() then
        ExecuteA64(0xD5130501<31:0>);
        // A64 "MSR DBGDTRTX_EL0,X1"
    else
        ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);
        // T32 "MSR DBGDTRTXint,R1"
    // "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;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return;  // Software lock locked: ignore write
DTRTX = value;
return;
Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ====================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
        EDSCR.RXfull = '0';
        return result;

Library pseudocode for shared/debug/dccanditr/DTR

bits(32) DTRRX;
bits(32) DTRTX;
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value

if EDPSR<6:5,0> != '001' then  // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;

if EDSCR.ERR == '1' then return;  // Error flag set: ignore write

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return;  // Software lock locked: ignore write

if !Halted() then return;  // Non-debug state: ignore write

if EDSCR.ITE == '0' || EDSCR.MA == '1' then
    EDSCR.ITO = '1';  EDSCR.ERR = '1';  // Overrun condition: block write
    return;

// ITE indicates whether the processor is ready to accept another instruction; the processor
// may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
// is no indication that the pipeline is empty (all instructions have completed). In this
// pseudocode, the assumption is that only one instruction can be executed at a time,
// meaning ITE acts like "InstrCompl".
EDSCR.ITE = '0';

if !UsingAArch32() then
    ExecuteA64(value);
else
    ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

EDSCR.ITE = '1';

return;
Library pseudocode for shared/debug/halting/DCPSInstruction
// DCPSInstruction()
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then UNDEFINED;
            else handle_el = EL1;
        when EL2
            if !HaveEL(EL2) then UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif !IsSecureEL2Enabled() && IsSecure() then UNDEFINED;
            else handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif !IsSecureEL2Enabled() && IsSecure() then UNDEFINED;
            else handle_el = EL3;
        when EL4
            if PSTATE.SP == EL4 && !IsSecure() then UNDEFINED;
        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
            else
                // Targeting AArch64
                if UsingAArch32() then
                    AArch64.MaybeZeroRegisterUppers();
                    MaybeZeroSVEUppers(target_el);
                    PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
                    if HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
                        (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                        HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0')) then
                        PSTATE.PAN = '1';
                        ELR[] = bits(64) UNKNOWN; SPSR[] = bits(64) UNKNOWN; ESPR[] = bits(64) UNKNOWN;
                        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
                        if HaveLADExt() 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(Unpredictable_IESEBinDebug) then
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() & 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(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    bits(64) spsr = SPSR[];
    SetPSTATEFromPSR(spsr);

    // PSTATE.{N,Z,C,V,Q,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(13) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1';  // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;

    UpdateEDSCRFields();  // Update EDSCR PE state flags
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 = '00001'; // Signal restarting
EDESR<2:0> = '000'; // Clear any pending Halting debug events

bits(64) new_pc;
bits(64) spsr;
if UsingAArch32() then
    new_pc = ZeroExtend(DLR);
spsr = ZeroExtend(DSPSR);
else
    new_pc = DLR_EL0;
spsr = DSPSR_EL0;
// If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
if UsingAArch32() then
    SetPSTATEFromPSR(spsr<31:0>); // Can update privileged bits, even at EL0
else
    SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0

if UsingAArch32() then
    if ConstrainUnpredictableBool(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' & ConstrainUnpredictableBool(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;
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr();
    bits(32) spsr_32;
    bits(64) spsr_64;
    if UsingAArch32() then
        spsr_32 = GetPSRFromPSTATE(DebugState);
    else
        spsr_64 = GetPSRFromPSTATE(DebugState);
    if (HaveBTIExt() &
        !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive, DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &
        ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
        if UsingAArch32() then
            spsr_32<11:10> = '00';
        else
            spsr_64<11:10> = '00';
    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr_32;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr_64;
    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        assert EDSCR.SDD == '1';                // Otherwise EDSCR.SDD is RES1
        EDSCR.MA = '0';
    // In Debug state:
    // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
    //     are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.{IT,T} are ignored.
    // * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.
    if UsingAArch32() then
        PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F>    = bits(6)  UNKNOWN;
        PSTATE.BTYPE = '00';
        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

// HaltOnBreakpointOrWatchpoint()
// =================-------------
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && ESCR.HDE == '1' && OSLR_EL1.OSLK == '0';

Library pseudocode for shared/debug/halting/Halted

// Halted()
// ========

boolean Halted()
    return !(ESCR.STATUS IN {'000001', '000010'});

Library pseudocode for shared/debug/halting/HaltingAllowed

// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elsif IsSecure() then
        return ExternalSecureInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();

Library pseudocode for shared/debug/halting/Restarting

// Restarting()
// ============

boolean Restarting()
    return ESCR.STATUS == '000001';

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

StopInstructionPrefetchAndEnableITR();
UpdateEDSCRFields()
// Update EDSCR PE state fields

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;

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

```c
// 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

```c
// 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

```c
// 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

```c
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```c
// 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(EL1) 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.
    // "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
        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;
// ExternalDebugInterruptsDisabled()
// --------------------------------------------
// Determine whether EDSCR disables interrupts routed to 'target'

boolean ExternalDebugInterruptsDisabled(bits(2) target)
if HaveV8p4Debug() then
  if target == EL3 || IsSecure() then
    int_dis = (EDSCR.INTdis[0] == '1' && ExternalSecureInvasiveDebugEnabled());
  else
    int_dis = (EDSCR.INTdis[0] == '1');
else
  case target of
    when EL3
      int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
    when EL2
      int_dis = (EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled());
    when EL1
      if IsSecure() then
        int_dis = (EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled());
      else
        int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
  return int_dis;

// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);

// In a simple sequential execution of the program, CreatePCSample is executed each time the PE
// executes an instruction that can be sampled. An implementation is not constrained such that
// reads of EDPCSRlo return the current values of PC, etc.
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<31:0>;
  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() || HaveV82Debug()) && !ELUsingAArch32(EL2) then
      pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
    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 "generate error response";
  return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0';  // Software locked: no side-effects

if pc_sample.valid then
  sample = pc_sample.pc<31:0>;
  if update then
    if HaveVirtHostExt() && EDSCR.SC2 == '1' then
      EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
      EDPCSRhi.EL = pc_sample.el;
      EDPCSRhi.NS = pc_sample.ns;
    else
      EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
      EDCIDSR = pc_sample.contextidr;
    end
    if (HaveVirtHostExt() || HaveV82Debug()) && EDSCR.SC2 == '1' then
      EDVIDSR = (if HaveEL(EL2) && pc_sample.ns == '1' then pc_sample.contextidr_el2
        else bits(32) UNKNOWN);
    else
      if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1, EL0} then
        EDVIDSR.VMID = pc_sample.vmid;
      else
        EDVIDSR.VMID = Zeros();
      end
      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");
    end
    sample = Ones(32);
  end
  if update then
    EDPCSRhi = bits(32) UNKNOWN;
    EDCIDS = bits(32) UNKNOWN;
    EDVIDSR = bits(32) UNKNOWN;
end

return sample;

Library pseudocode for shared/debug/samplebasedprofiling/PCSample

type PCSample is (
  boolean valid,
  bits(64) pc,
  bits(2) el,
  bit rw,
  bit ns,
  boolean has_el2,
  bits(32) contextidr,
  bits(32) contextidr_el2,
  boolean el0h,
  bits(16) vmid
)

PCSample pc_sample;
Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

// PMPCSR{} (read)
// ===============

bits(32) PMPCSR[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response"
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || PMLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
        PMPCSR.EL = pc_sample.el;
        PMPCSR.NS = pc_sample.ns;
        PMCID1SR = pc_sample.contextidr;
        PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;
        PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1, EL0} && !pc_sample.el0h
            then pc_sample.vmid else bits(16) UNKNOWN);
    else
        sample = Ones(32);
        if update then
            PMPCSR<55:32> = bits(24) UNKNOWN;
            PMPCSR.EL = bits(2) UNKNOWN;
            PMPCSR.NS = bit UNKNOWN;
            PMCID1SR = bits(32) UNKNOWN;
            PMCID2SR = bits(32) UNKNOWN;
            PMVIDSR.VMID = bits(16) UNKNOWN;
    end

return sample;

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

// CheckSoftwareStep()
// ===================

// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

// Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
// AArch32 state. However, because Software Step is only active when the debug target Exception
// level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
step_enabled = !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() && MDSCR_EL1.SS ==
if step_enabled && PSTATE.SS == '0' then
    AArch64.SoftwareStepException();
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    if UsingAArch32() then
        assert N == 32;
        assert N == 64;
    assert Halted() || Restarting() || PSTATE.EL != EL0;
    if Restarting() then
        enabled_at_source = FALSE;
    elseif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();
    if IllegalExceptionReturn(spsr) then
        dest = PSTATE.EL;
    else
        (valid, dest) = ELFromSPSR(spsr);  assert valid;
        dest_is_secure = IsSecureBelowEL3() || dest == EL3;
        dest_using_32 = (if dest == EL0 then spsr<4> == '1' else ELUsingAArch32(dest));
        if dest_using_32 then
            enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, dest_is_secure);
        else
            mask = spsr<9>;
            enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, dest_is_secure, mask);
        ELD = DebugTargetFrom(dest_is_secure);
        if !ELUsingAArch32(ELD) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
            SS_bit = spsr<21>;
        else
            SS_bit = '0';
    return SS_bit;

Library pseudocode for shared/debug/softwarestep/SSAdvance

// SSAdvance()
// ==========
// Advance the Software Step state machine.

SSAdvance()
    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';
    if active_not_pending then PSTATE.SS = '0';
    return;
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 or LDAEX that failed its condition code test.
// Otherwise returns TRUE if the instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();

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_ESRCOND_PASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else
            // 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;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';
    return syndrome;
Library pseudocode for shared/exceptions/exceptions/Exception

```
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_CP14RTTrap,            // Trapped AArch32 MCR or MRC access to CP14
    Exception_AdvSIMDFPAccessTrap,   // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap,              // Trapped access to SIMD or FP ID register
    Exception_LDST64BTrap,           // Trapped access to ST64BV, ST64BV0, ST64B and LD64B
    Exception_PACTrap,               // Trapped invalid PAC use
    Exception_CP14RTTTrap,           // Trapped MRRC access to CP14 from AArch32
    Exception_IllegalState,         // Illegal Execution state
    Exception_SupervisorCall,        // Supervisor Call
    Exception_HypervisorCall,        // Hypervisor Call
    Exception_MonitorCall,           // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap,    // Trapped MRS or MSR system register access
    Exception_ERetTrap,              // Trapped invalid ERET use
    Exception_InstructionAbort,      // Instruction Abort or Prefetch Abort
    Exception_PCAAlignment,          // PC alignment fault
    Exception_DataAbort,             // Data Abort
    Exception_NV2DataAbort,          // Data abort at EL1 reported as being from EL2
    Exception_PACFail,               // PAC Authentication failure
    Exception_SPIAlignment,          // SP alignment fault
    Exception_FPTrappedException,    // IEEE trapped FP exception
    Exception_SError,                // SErrror 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_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(5) syndrome2,       // ST64BV(0) return value register specifier
                        bits(64) vaddress,       // Virtual fault address
                        boolean ivalid,          // 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.syndrome2 = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros();

    return r;

Library pseudocode for shared/exceptions/traps/ReservedValue

// ReservedValue()
// ===============
ReservedValue()

    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();

Library pseudocode for shared/exceptions/traps/UnallocatedEncoding

// UnallocatedEncoding()
// ===============
UnallocatedEncoding()

    if UsingAArch32() && AArch32.ExecutingCP10or11Instr() then
        FPEXC.DEX = '0';
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
Library pseudocode for shared/functions/aborts/EncodeLDFSC

// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)

  bits(6) result;
  case statuscode of
    when Fault_AddressSize result = '0000':level<1:0>; assert level IN {0,1,2,3};
    when Fault_AccessFlag result = '0010':level<1:0>; assert level IN {1,2,3};
    when Fault_Permission result = '0011':level<1:0>; assert level IN {1,2,3};
    when Fault_Translation result = '0001':level<1:0>; assert level IN {0,1,2,3};
    when Fault_SyncExternal result = '010000';
    when Fault_SyncExternalOnWalk result = '0101':level<1:0>; assert level IN {0,1,2,3};
    when Fault_SyncParity result = '011000';
    when Fault_SyncParityOnWalk result = '0111':level<1:0>; assert level IN {0,1,2,3};
    when Fault_AsyncParity result = '011001';
    when Fault_AsyncExternal result = '010001';
    when Fault_AsyncParity result = '010011';
    when Fault_AsyncExternal result = '010001';
    when Fault_AsyncParity result = '010011';
    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 unreachable();

  return result;

Library pseudocode for shared/functions/aborts/IPAValid

// IPAValid()
// =========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)

  assert fault.statuscode != Fault_None;
  if fault.s2fs1walk then
    return fault.statuscode IN {Fault_AccessFlag, Fault_Permission, Fault_Translation, Fault_AddressSize};
  elsif fault.secondstage then
    return fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
  else
    return FALSE;

Library pseudocode for shared/functions/aborts/IsAsyncAbort

// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE otherwise.

boolean IsAsyncAbort(Fault statuscode)

  assert statuscode != Fault_None;
  return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ===============

boolean IsAsyncAbort(FaultRecord fault)

  return IsAsyncAbort(fault.statuscode);
Library pseudocode for shared/functions/aborts/IsDebugException

// IsDebugException()
// =============

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;

Library pseudocode for shared/functions/aborts/IsExternalAbort

// IsExternalAbort()
// ===============

// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk, Fault_AsyncExternal, Fault_AsyncParity});

// IsExternalAbort()
// ===============

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode);

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

// IsExternalSyncAbort()
// ====================

// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk});

// IsExternalSyncAbort()
// ====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode);

Library pseudocode for shared/functions/aborts/IsFault

// IsFault()
// =========

// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;
Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

// IsSErrorInterrupt()
// ===============
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
  assert statuscode != Fault_None;
  return (statuscode IN {Fault.AsyncExternal, Fault.AsyncParity});

Library pseudocode for shared/functions/aborts/IsSecondStage

// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
  assert fault.statuscode != Fault_None;
  return fault.secondstage;

Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

// Returns the extended syndrome information for a second stage fault.
// <10> - Syndrome valid bit. The syndrome is only valid for certain types of access instruction.
// <9:8> - Access size.
// <7> - Sign extended (for loads).
// <6:2> - Transfer register.
// <1> - Transfer register is 64-bit.
// <0> - Instruction has acquire/release semantics.
bits(11) LSInstructionSyndrome();

Library pseudocode for shared/functions/common/ASR

// ASR()
// ======

bits(N) ASR(bits(N) x, integer shift)
  assert shift >= 0;
  if shift == 0 then
    result = x;
  else
    (result, -) = ASR_C(x, shift);
  return result;

Library pseudocode for shared/functions/common/ASR_C

// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
  assert shift > 0;
  extended_x = SignExtend(x, shift+N);
  result = extended_x<shift+N-1: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;

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()
// ===============

to 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

```java
// IsOnes()
// ========
boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

Library pseudocode for shared/functions/common/IsZero

```java
// IsZero()
// ========
boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

Library pseudocode for shared/functions/common/IsZeroBit

```java
// IsZeroBit()
// ===========
bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

Library pseudocode for shared/functions/common/LSL

```java
// 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

```java
// 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

```java
// LSR()
// =====
bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```
Library pseudocode for shared/functions/common/LSR_C

// LSR_C()
// ========

(bits(N), bit) LSR_C(bits(N) x, integer shift)
assert shift > 0;
extended x = ZeroExtend(x, shift+N);
result = extended_x<shift+N:shift>;
carry_out = extended_x<shift-1>;
return (result, carry_out);

Library pseudocode for shared/functions/common/LowestSetBit

// LowestSetBit()
// =============

integer LowestSetBit(bits(N) x)
for i = 0 to N-1
  if x<i> == '1' then return i;
return N;

Library pseudocode for shared/functions/common/Max

// Max()
// =====

integer Max(integer a, integer b)
  return if a >= b then a else b;
// Max()
// =====

real Max(real a, real b)
  return if a >= b then a else b;

Library pseudocode for shared/functions/common/Min

// Min()
// =====

integer Min(integer a, integer b)
  return if a <= b then a else b;
// Min()
// =====

real Min(real a, real b)
  return if a <= b then a else b;

Library pseudocode for shared/functions/common/Ones

// Ones()
// ======

bits(N) Ones(integer N)
  return Replicate('1',N);
// Ones()
// ======

bits(N) Ones()
  return Ones(N);
Library pseudocode for shared/functions/common/ROR

// ROR()
// ======

bits(N) ROR(bits(N) x, integer shift)
assert shift >= 0;
if shift == 0 then
    result = x;
else
    (result, -) = ROR_C(x, shift);
return result;

Library pseudocode for shared/functions/common/ROR_C

// ROR_C()
// ========

(bits(N), bit) ROR_C(bits(N) x, integer shift)
assert shift != 0;
m = shift MOD N;
result = LSR(x,m) OR LSL(x,N-m);
carry_out = result<^_N-1>;
return (result, carry_out);

Library pseudocode for shared/functions/common/Replicate

// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
assert N MOD M == 0;
return Replicate(x, N DIV M);

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

```plaintext
// SignExtend()
// ============

bits(N) SignExtend<bits(M) x, integer N)  
    assert N >= M;  
    return Replicate(x<N-1>, N-M) : x;

// SignExtend()
// ============

bits(N) SignExtend<bits(M) x)  
    return SignExtend(x, N);
```

Library pseudocode for shared/functions/common/UInt

```plaintext
// 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

```plaintext
// 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

```plaintext
// Zeros()
// ======

bits(N) Zeros(integer N)  
    return Replicate('0',N);

// Zeros()
// ======

bits(N) Zeros()  
    return Zeros(N);
```

Library pseudocode for shared/functions/crc/BitReverse

```plaintext
// BitReverse()
// ============

bits(N) BitReverse<bits(N) data)  
    bits(N) result;  
    for i = 0 to N-1  
        result<N-i-1> = data<i>;  
    return result;
```
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";

// 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>;

// AESInvMixColumns()
// ===============

// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR FFmul09(in3<c*8+:8>);
        out1<c*8+:8> = FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR FFmul0D(in3<c*8+:8>);
        out2<c*8+:8> = FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR FFmul0E(in3<c*8+:8>);
        out3<c*8+:8> = FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR FFmul0E(in3<c*8+:8>);

    return (out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> : out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> : out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> : out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>);

// AESInvShiftRows()
// ================

// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (op< 24+:8> : op< 48+:8> : op< 72+:8> : op< 96+:8> : op<120+:8> : op< 16+:8> : op< 40+:8> : op< 64+:8> : op< 88+:8> : op<112+:8> : op<  8+:8> : op< 32+:8> : op< 56+:8> : op< 80+:8> : op<104+:8> : op<  0+:8>);
Library pseudocode for shared/functions/crypto/AESInvSubBytes

// AESSubBytes()  
// ==============  
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESSubBytes(bits(128) op)
    // Inverse S-box values
    bits(16*8) GF2_inv = {
        /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :  
        /*E*/ 0x61995383cbbec8b0f52aee4d3be0a0<127:0> :  
        /*D*/ 0xef9cc939f7ae52d0d4ab519a97f5160<127:0> :  
        /*C*/ 0x5f6ec8027591012b131c707833a8dd1f<127:0> :  
        /*B*/ 0xf45acd78fec0db9a2079dc64be3e56fc<127:0> :  
        /*A*/ 0x1b6be18aa0e2b76f89c5291d71af147<127:0> :  
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :  
        /*8*/ 0x736eb4f0ceccff297eac6d74f4111913a<127:0> :  
        /*7*/ 0x6b6a130103dbaf1020f3ca8f1e2cdd0<127:0> :  
        /*6*/ 0x0645b3b80558e4f70ad3bc8c0ab990<127:0> :  
        /*5*/ 0x849d8da75746155edab9edfd5048706<127:0> :  
        /*4*/ 0x92b6655dca4d4169b688646f872<127:0> :  
        /*3*/ 0x25d1b86d49a2576b224d92866a1e08<127:0> :  
        /*2*/ 0xe4c3fa20b954ce3d2c2a632947b54<127:0> :  
        /*1*/ 0x0ce9dce44443e8347f2f9b239e37c<127:0> :  
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0> }
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
    return out;

Library pseudocode for shared/functions/crypto/AESMixColumns

// AESMixColumns()  
// ===============  
// Transformation in the Cipher that takes all of the columns of the 
// State and mixes their data (independently of one another) to 
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
    bits(32) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(32) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(32) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(32) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(32) out0;
    bits(32) out1;
    bits(32) out2;
    bits(32) out3;
    for c = 0 to 3
        out0<0*8+:8> = FFmul02(in0<0*8+:8>) EOR FFmul03(in1<0*8+:8>) EOR in2<0*8+:8> EOR in3<0*8+:8>;
        out1<0*8+:8> = in0<0*8+:8> EOR FFmul02(in1<0*8+:8>) EOR FFmul03(in2<0*8+:8>) EOR FFmul04(in3<0*8+:8>);
        out2<0*8+:8> = in0<0*8+:8> EOR in1<0*8+:8> EOR FFmul02(in2<0*8+:8>) EOR FFmul03(in3<0*8+:8>);
        out3<0*8+:8> = FFmul03(in0<0*8+:8>) EOR in1<0*8+:8> EOR in2<0*8+:8> EOR FFmul04(in3<0*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       */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b48ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610f634866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6ba41c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65ef4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162ac3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee46b8902a22d4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6c4533f5d928f40a351<127:0> :
        /*6*/ 0x89f3c507f2f458534d4fbaaeaf0<127:0> :
        /*5*/ 0xcf584c4a39bce6a5bb1fc20ed0d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a06e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebea28012079a859618c323c704<127:0> :
        /*2*/ 0x1531d871fe5a534ccf73f362693f9d7<127:0> :
        /*1*/ 0xc072a49c6afda4af04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b70138c56f6b27b77c63<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 2733


Library pseudocode for shared/functions/crypto/FFmul09

/* FFmul09() */
bits(8) FFmul09(bits(8) b)
bits(256*8) FFmul_09 = (***)

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 = (***)

return FFmul_0B<UInt(b)*8+:8>;

Shared Pseudocode Functions
Library pseudocode for shared/functions/crypto/FFmul0D

```c
// FFmul0D()
// ========

// 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*/ 0x979A8D80A3AEB9B4FFF2E58CB6D1DC<127:0> 0
//          */E*/ 0x47A45D50737E69642F2235381B16010C<127:0> 0
//          */D*/ 0x2C2136381B15202F44955E3707D6A67<127:0> 0
//          */C*/ 0xFCF16E6C8C5C02DF949F83A0ADBBAB7<127:0> 0
//          */B*/ 0xFAC7E80DECE3D49229F88B5A6BBCB1<127:0> 0
//          */A*/ 0x2A273D31E130409244F5B5557B6C1<127:0> 0
//          */9*/ 0x14C55B675786F22292433E1D10070A<127:0> 0
//          */8*/ 0x919CBB86A588BF2F94E4C3EDC0D7DA<127:0> 0
//          */7*/ 0x4D04D57A9794636E25283F3211C0806<127:0> 0
//          */6*/ 0x9D90B7AA9A43BE5F8EFE2CCDB6D<127:0> 0
//          */5*/ 0x6FBD8C1E26F0D59E98489AA7B0BD<127:0> 0
//          */4*/ 0x262B3C1121F08054E345597A77606D<127:0> 0
//          */3*/ 0x202D3A714190E034845525F7C71666B<127:0> 0
//          */2*/ 0xF0FDEAE74C90DE39958B28FAC0A1B6BB<127:0> 0
//          */1*/ 0x98B618B4F2A83B58F33E9E4C7ADDD<127:0> 0
//          */0*/ 0x4B46515C7F7265823E3991A0D00<127:0> 0
//        )
//    return FFmul_0D<UInt>(b)*8+:8);
```

Library pseudocode for shared/functions/crypto/FFmul0E

```c
// FFmul0E()
// ========

// 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*/ 0x8D83919F8B58A97DF3E1EC5CB0907<127:0> 0
//          */E*/ 0x6D63717F555B49471D13010F25283937<127:0> 0
//          */D*/ 0x56584A446E0727C26283A341E10020C<127:0> 0
//          */C*/ 0x86BBAA48E89209C6C5B0A4DFE0E2EC<127:0> 0
//          */B*/ 0x202E3C32181604A05E442686747A<127:0> 0
//          */A*/ 0xC0C8C46C0F905B8B3863C9F9C99<127:0> 0
//          */9*/ 0x17190B052F2133367697B755F15434D<127:0> 0
//          */8*/ 0xF7F9EBE5CFE10D3D8799B958BF1A3AD<127:0> 0
//          */7*/ 0x616F7D35957454B111F0D32927353B<127:0> 0
//          */6*/ 0x818F9D9B87A5ABF1FFDE3C9C7D5DB<127:0> 0
//          */5*/ 0x98B4A68828C9E90C40D6802FCEEE<127:0> 0
//          */4*/ 0x5A544648626C7E702A243638121C0E00<127:0> 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";
Library pseudocode for shared/functions/crypto/HaveSM3Ext

```plaintext
// 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";
```

Library pseudocode for shared/functions/crypto/HaveSM4Ext

```plaintext
// 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";
```

Library pseudocode for shared/functions/crypto/ROL

```plaintext
// 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);
```

Library pseudocode for shared/functions/crypto/SHA256hash

```plaintext
// 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);
```

Library pseudocode for shared/functions/crypto/SHAchoose

```plaintext
// 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

```plaintext
// 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

```plaintext
// 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

```plaintext
// 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

```plaintext
// 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

```plaintext
// Sbox()
// ======

// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = 0xd690e9feccel3db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491e<2047:0>;
    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```plaintext
// 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

```plaintext
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```
// ClearExclusiveMonitors()
// ========================
// Clear the local Exclusives monitor for the executing PE.
ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());

// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();

// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);

// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);

// Return the ID of the currently executing PE.
integer ProcessorID();

// AArch32.HaveHPDExt()
// ====================
boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);

// AArch64.HaveHPDExt()
// ====================
boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
Library pseudocode for shared/functions/extension/Have52BitIPAAndPASpaceExt

// Have52BitIPAAndPASpaceExt()
// ===========================
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
return (HasArchVersion(ARMv8p7) &&
    boolean IMPLEMENTATION_DEFINED "Has 52-bit IPA and PA support" &&
    Have52BitVAExt() && Have52BitPAExt());

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/HaveAltFP

// HaveAltFP()
// ===========
// Returns TRUE if alternative Floating-point extension support is implemented, and FALSE otherwise.

boolean HaveAltFP()
return HasArchVersion(ARMv8p7);
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/HaveDGHEExt

// HaveDGHEExt()
// =============
// Returns TRUE if Data Gathering Hint instruction support is implemented, and FALSE otherwise.
boolean HaveDGHEExt()
return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";

Library pseudocode for shared/functions/extension/HaveDITExt

// HaveDITExt()
// =============
boolean HaveDITExt()
return HasArchVersion(ARMv8p4);
// HaveDOTPExt()
// =============
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.

boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p4) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has Dot Product extension");

// HaveDoPD()
// =========
// Returns TRUE if Debug Over Power Down extension
// support is implemented and FALSE otherwise.

boolean HaveDoPD()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";

// HaveDoubleFaultExt()
// ====================

boolean HaveDoubleFaultExt()
    return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB());

// HaveDoubleLock()
// ================
// Returns TRUE if support for the OS Double Lock is implemented.

boolean HaveDoubleLock()
    return !HasArchVersion(ARMv8p4) || boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";

// HaveE0PDEExt()
// =============
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveE0PDEExt()
    return HasArchVersion(ARMv8p5);

// HaveECVExt()
// ============
// Returns TRUE if Enhanced Counter Virtualization extension
// support is implemented, and FALSE otherwise.

boolean HaveECVExt()
    return HasArchVersion(ARMv8p6);
// HaveEMPAMExt()
// ==============
// Returns TRUE if Enhanced MPAM is implemented, and FALSE otherwise.

boolean HaveEMPAMExt() {
  return HasArchVersion(ARMv8p6) && HaveEMPAMExt() &&
  boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension");
}

// HaveExtendedCacheSets()
// =======================

boolean HaveExtendedCacheSets() {
  return HasArchVersion(ARMv8p3);
}

// HaveExtendedECDebugEvents()
// ===========================

boolean HaveExtendedECDebugEvents() {
  return HasArchVersion(ARMv8p2);
}

// HaveExtendedExecuteNeverExt()
// =============================

boolean HaveExtendedExecuteNeverExt() {
  return HasArchVersion(ARMv8p2);
}

// HaveFCADDExt()
// ==============

boolean HaveFCADDExt() {
  return HasArchVersion(ARMv8p3);
}

// HaveFGTExt()
// ============

boolean HaveFGTExt() {
  return HasArchVersion(ARMv8p6) && !ELUsingAArch32(EL2);
}

// HaveFJCVTZSExt()
// ================

boolean HaveFJCVTZSExt() {
  return HasArchVersion(ARMv8p3);
}
Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```plaintext
// HaveFP16MulNoRoundingToFP32Ext()
// ===================================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions,
// and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
if !HaveFP16Ext() then return FALSE;
if HasArchVersion(ARMv8p4) then return TRUE;
return (HasArchVersion(ARMv8p2) &&
    boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

Library pseudocode for shared/functions/extension/HaveFeatLS64

```plaintext
// HaveFeatLS64()
// ==============
// Returns TRUE if the LD64B, ST64B, ST64BV, and ST64BV0 instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
return (HasArchVersion(ARMv8p7) &&
    boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");
```

Library pseudocode for shared/functions/extension/HaveFeatRPRES

```plaintext
// HaveFeatRPRES()
// ===============
// Returns TRUE if the Reciprocal Estimate and Reciprocal Square Root Estimate
// instructions have increased precision, and FALSE otherwise.

boolean HaveFeatRPRES()
return (HasArchVersion(ARMv8p7) &&
    boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate precision support"
     HaveAltFP());
```

Library pseudocode for shared/functions/extension/HaveFeatWFxT

```plaintext
// HaveFeatWFxT()
// ==============
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatWFxT()
return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveFeatXS

```plaintext
// HaveFeatXS()
// ============
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
// are supported, and FALSE otherwise.

boolean HaveFeatXS()
return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveFlagFormatExt

```plaintext
// HaveFlagFormatExt()
// ===================
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFlagFormatExt()
return HasArchVersion(ARMv8p5);
```
// HaveFlagManipulateExt()
// =======================
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()

    return HasArchVersion(ARMv8p4);

// HaveFrintExt()
// ==============
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()

    return HasArchVersion(ARMv8p5);

// HaveHCRXEL2Ext()
// ================
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveHCRXEL2Ext()

    return HasArchVersion(ARMv8p7);

// HaveHPMDExt()
// =============

boolean HaveHPMDExt()

    return HasArchVersion(ARMv8p1);

// HaveIDSExt()
// ============

boolean HaveIDSExt()

    return HasArchVersion(ARMv8p4);

// HaveIESB()
// ===========

boolean HaveIESB()

    return (HaveRASExt() &&
    boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");

// HaveInt8MatMulExt()
// ===================
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()

    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch64 Int8 Mat Mul extension");
// HaveLSE2Ext()
// =============
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
    return HasArchVersion(ARMv8p4);

// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has MPAM extension");

// HaveMTE2Ext()
// =============
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.

boolean HaveMTE2Ext()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension";

// HaveMTE3Ext()
// =============
// Returns TRUE if MTE Asymmetric Fault Handling support is implemented, and FALSE otherwise.

boolean HaveMTE3Ext()
    return ((HasArchVersion(ARMv8p7) && HaveMTE2Ext()) || (HasArchVersion(ARMv8p5) &&
              boolean IMPLEMENTATION_DEFINED "Has MTE3 extension");

// HaveMTEEExt()
// =============
// Returns TRUE if MTE implemented, and FALSE otherwise.

boolean HaveMTEEExt()
    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/HavePAN3Ext

```java
// HavePAN3Ext()
// =============
// Returns TRUE if SCTLR_EL1.EPAN and SCTLR_EL2.EPAN support is implemented,
// and FALSE otherwise.

boolean HavePAN3Ext()
    return HasArchVersion(ARMv8p7) || (HasArchVersion(ARMv8p1) &&
        boolean IMPLEMENTATION_DEFINED "Has PAN3 extension");
```

Library pseudocode for shared/functions/extension/HavePANExt

```java
// HavePANExt()
// ============

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HavePMUv3p7

```java
// HavePMUv3p7()
// =============
// Returns TRUE if the PMUv3p7 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p7()
    return (HasArchVersion(ARMv8p7) && Havev85PMU() &&
        boolean IMPLEMENTATION_DEFINED "Has PMUv3p7 extension");
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```java
// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```
Library pseudocode for shared/functions/extension/HavePrivATExt

    // HavePrivATExt()
    // ===============
    boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

    // HaveQRDMLAHExt()
    // ================
    boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);
    boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);
    boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);

Library pseudocode for shared/functions/extension/HaveRASExt

    // HaveRASExt()
    // ===========
    boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) || boolean IMPLEMENTATION_DEFINED "Has RAS extension");

Library pseudocode for shared/functions/extension/HaveRNG

    // HaveRNG()
    // =========
    // Returns TRUE if Random Number Generator extension
    // support is implemented and FALSE otherwise.
    boolean HaveRNG()
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension";

Library pseudocode for shared/functions/extension/HaveSBExt

    // HaveSBExt()
    // ===========
    // Returns TRUE if support for SB is implemented, and FALSE otherwise.
    boolean HaveSBExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";

Library pseudocode for shared/functions/extension/HaveSSBSExt

    // HaveSSBSExt()
    // =============
    // Returns TRUE if support for SSBS is implemented, and FALSE otherwise.
    boolean HaveSSBSExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
// HaveSecureEL2Ext()
// ================
// Returns TRUE if Secure EL2 is implemented.
boolean HaveSecureEL2Ext()
    return HasArchVersion(ARMv8p4);

// HaveSecureExtDebugView()
// =========================
// Returns TRUE if support for Secure and Non-secure views of debug peripherals is implemented.
boolean HaveSecureExtDebugView()
    return HasArchVersion(ARMv8p4);

// HaveSelfHostedTrace()
// =====================
boolean HaveSelfHostedTrace()
    return HasArchVersion(ARMv8p4);

// HaveSmallTranslationTblExt()
// ============================
// Returns TRUE if Small Translation Table Support is implemented.
boolean HaveSmallTranslationTableExt()
    return HasArchVersion(ARMv8p4) && boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension";

// HaveStage2MemAttrControl()
// ===========================
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is implemented.
boolean HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4);

// HaveStatisticalProfiling()
// ==========================  
// Returns TRUE if Statistical Profiling Extension is implemented, and FALSE otherwise.
boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);

// HaveStatisticalProfilingv1p1()
// ==============================  
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.
boolean HaveStatisticalProfilingv1p1()
    return (HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");
// HaveStatisticalProfilingv1p2()
// ==============================
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p2()
    return (HasArchVersion(ARMv8p7) && HaveStatisticalProfiling() &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension");

// HaveTWEDExt()
// =============
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented, and FALSE otherwise.

boolean HaveTWEDExt()
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";

// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";

// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);

// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);

// HaveV82Debug()
// ==============

boolean HaveV82Debug()
    return HasArchVersion(ARMv8p2);

// HaveVirtHostExt()
// =================

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
Library pseudocode for shared/functions/extension/Havev85PMU

```plaintext
// Havev85PMU()
// ============
// Returns TRUE if v8.5-Performance Monitor Unit extension
// support is implemented, and FALSE otherwise.

boolean Havev85PMU()
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has PMUv3p5 extension";
```

Library pseudocode for shared/functions/extension/Havev8p4Debug

```plaintext
// Havev8p4Debug()
// ===============
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.

boolean Havev8p4Debug()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```plaintext
// 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

```plaintext
// 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;
```

Shared Pseudocode Functions
Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

```c
// 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 = BF Mul(Elem[op1, 4*i + 2*k + 0, 16], Elem[op2, 4*j + 2*k + 1, 16]);
        prod1 = BF Mul(Elem[op1, 4*i + 2*k + 1, 16], Elem[op2, 4*j + 2*k + 0, 16]);
    sum = BF Add(sum, BF Add(prod0, prod1));
Elem[result, 2*i + j, 32] = sum;

return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMul

```c
// BF Mul()
// ========
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BF Mul(bits(16) op1, bits(16) op2)

bits(32) result;

(type1,sign1,value1) = BF Unpack(op1);
(type2,sign2,value2) = BF Unpack(op2);
if type1 == FPType_QNaN || type2 == FPType_QNaN then
    result = FP DefaultNaN();
else
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);
    if (inf1 && zero2) || (zero1 && inf2) then
        result = FP DefaultNaN();
    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;
```

Library pseudocode for shared/functions/float/bfloat/BFMulAdd

```c
// BF MulAdd()
// ===========
// Used by BFMLALB and BFMLALT instructions.

bits(N) BF MulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altpf = HaveAltFP() && fpcr.AH == '1'; // When TRUE:
    boolean fpexc = !altpf; // Do not generate floating point exceptions
if altpf then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
if altpf then fpcr.RMode = '00'; // Use RNE rounding mode
return BF MulAdd(addend, op1, op2, fpcr, fpexc);
```
Library pseudocode for shared/functions/float/bfloat/BFRound

// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)
    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126;  E = 8;  F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign);  // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

    return result;}
Library pseudocode for shared/functions/float/bfloat/BFUnpack

// BFUnpack()
// ============
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)
  assert N IN {16,32};
  if N == 16 then
    sign   = fpval<15>;
    exp    = fpval<14:7>:
    frac   = fpval<6:0> : Zeros(16);
  else  // N == 32
    sign   = fpval<31>;
    exp    = fpval<30:23>;
    frac   = fpval<22:0>;
  if IsZero(exp) then
    fptype = FPType_Zero;  value = 0.0;    // Fixed Flush to Zero
  elsif IsOnes(exp) then
    if IsZero(frac) then
      fptype = FPType_Infinity;  value = 2.0^1000000;
    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);
  if sign == '1' then value = -value;
  return (fptype, sign, value);
Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```c
int FPConvertBF(int op, int fpcr, int rounding)
{
    int result;                                // BF16 value in top 16 bits
    int altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    int fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then rounding = FPRounding_TIEEVEN;    // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    if fpitype == FPTYPE_SNaN || fpitype == FPTYPE_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
    if fpitype == FPTYPE_SNaN then
        if fpexc then
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif fpitype == FPTYPE_INFINITY then
            result = FPInfinity(sign);
        elsif fpitype == FPTYPE_ZERO then
            result = FPZero(sign);
        else
            result = FPRoundCVBF(value, fpcr, rounding, fpexc);
    // Returns correctly rounded BF16 value from top 16 bits
    return result<31:16>;
}
```

Library pseudocode for shared/functions/float/bfloat/FPRoundCVBF

```c
int FPRoundCVBF(int op, int fpcr, int rounding, boolean fpexc)
{
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);
}
```
Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

// FixedToFP()
// ===========

// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and Rounding.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
assert N IN {16,32,64};
assert M IN {16,32,64};
bits(N) result;
assert fbits >= 0;
assert rounding != FPRounding_ODD;

// Correct signed-ness
int_operand = Int(op, unsigned);

// Scale by fractional bits and generate a real value
real_operand = Real(int_operand) / 2.0^fbits;

if real_operand == 0.0 then
    result = FPZero('0');
else
    result = FPRound(real_operand, fpcr, rounding);
return result;

Library pseudocode for shared/functions/float/fpabs/FPAbs

// FPAbs()
// ========

bits(N) FPAbs(bits(N) op)
assert N IN {16,32,64};
if !UsingAArch32() & HaveAltFP() then
    FPCRType fpcr = FPCR[];
if fpcr.AH == '1' then
    (ftype, -, -) = FPUnpack(op, fpcr, FALSE);
if ftype IN {FPType_SNan, FPType_QNan} then
    return op; // When fpcr.AH=1, sign of NaN has no consequence
return '0' : op<N-2:0>;
Library pseudocode for shared/functions/float/fpadd/FPAdd

```c
// 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();
    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);
        endif
    endif
else
    result = FPProcessDenorms(type1, type2, N, fpcr);
endif
return result;
```

Library pseudocode for shared/functions/float/fpcompare/FPCompare

```c
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
    result = '0011';
    if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';
        endif
    endif
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    if value1 == value2 then
        result = '0011';
    elsif value1 < value2 then
        result = '0001';
    else // value1 > value2
        result = '0010';
    endif
endif
return result;
```
Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

```c
// FPCompareEQ()
// ===========

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
result = FALSE;
if type1 == FPType_SNaN || type2 == FPType_SNaN then
FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 == value2);

FPProcessDenorms(type1, type2, N, fpcr);
return result;
```

Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```c
// FPCompareGE()
// ============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
result = FALSE;
FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 >= value2);

FPProcessDenorms(type1, type2, N, fpcr);
return result;
```

Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```c
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
result = FALSE;
FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 > value2);

FPProcessDenorms(type1, type2, N, fpcr);
return result;
```
Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========
// Convert floating point OP with N-bit precision to M-bit precision, // with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for // half-precision data ignores FZ16, but observes AHP.
bits(M) FPConvert(bits(N) op, FPCRTyp 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 == FPTyp_5NaN || fptype == FPTyp_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDa...
Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)
assert N IN {16,32,64};
assert M IN {16,32,64};
bits(M) result;
bits(51) frac;

sign = op<N-1>;

// Unpack payload from input NaN
case N of
  when 64 frac = op<50:0>;
  when 32 frac = op<21:0>: Zeros(29);
  when 16 frac = op<8:0>: Zeros(42);

// Repack payload into output NaN, while
// converting an SNaN to a QNaN.
case M of
  when 64 result = sign: Ones(M-52):frac;
  when 32 result = sign: Ones(M-23):frac<50:29>;
  when 16 result = sign: Ones(M-10):frac<50:42>;

return result;

Library pseudocode for shared/functions/float/fpcrtype/FPCRType

type FPCRType;

Library pseudocode for shared/functions/float/fpdecoderming/FPDecodeRM

// FPDecodeRM()
// ============
// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
  case rm of
    when '00' result = FPRounding_TIEAWAY; // A
    when '01' result = FPRounding_TIEEVEN; // N
    when '10' result = FPRounding_POSINF; // P
    when '11' result = FPRounding_NEGINF; // M

return result;

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

// FPDecodeRounding()
// ==================
// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
case rmode of
  when '00' return FPRounding_TIEEVEN; // N
  when '01' return FPRounding_POSINF; // P
  when '10' return FPRounding_NEGINF; // M
  when '11' return FPRounding_ZERO; // Z
// FPDefaultNaN()
// ===============

bits(N) FPDefaultNaN()
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
bit sign;
if !UsingAArch32() && HaveAltFP() then
    FPCRType fpcr = FPCR[];
sign = if fpcr.AH == '1' then '1' else '0';
else
    sign = '0';

bits(E) exp = Ones(E);
bites(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();
    elsif inf1 || zero2 then
        result = FPInfinity(sign1 EOR sign2);
    if !inf1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif zero1 || inf2 then
        result = FPZero(sign1 EOR sign2);
    else
        result = FPRound(value1/value2, fpcr);
if !zero2 then
    FPProcessDenorms(type1, type2, N, fpcr);
return result;

Library pseudocode for shared/functions/float/fpexc/FPExc

enumeration FPExc {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                  FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```c
// FPInfinity()
// ============
// bits(N) FPInfinity(bit sign)
// assert N IN {16,32,64};
// constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
// constant integer F = N - (E + 1);
// bits(E) exp = Ones(E);
// bits(F) frac = Zeros(F);
// return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```c
// FPMatMulAdd()
// =============
// // Floating point matrix multiply and add to same precision matrix
// // result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])
// bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRType fpcr)
// assert N == esize * 2 * 2;
// bits(N) result;
// bits(esize) prod0, prod1, sum;
// for i = 0 to 1
// for j = 0 to 1
// sum = Elem[addend, 2*i + j, esize];
// prod0 = FPMul(Elem[op1, 2*i + 0, esize],
//               Elem[op2, 2*j + 0, esize], fpcr);
// prod1 = FPMul(Elem[op1, 2*i + 1, esize],
//               Elem[op2, 2*j + 1, esize], fpcr);
// sum = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
// Elem[result, 2*i + j, esize] = sum;
// return result;
```
Library pseudocode for shared/functions/float/fpmax/FPMax

// FPMax()
// ========

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMax(op1, op2, fpcr, altfp);

// FPMax()
// ========
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behaviour.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean altfp)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if (altfp && type1 == FPType_Zero && type2 == FPType_Zero &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);
    if !done then
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';
                result = FPRound(value, fpcr, rounding, TRUE);
            FPProcessDenorms(type1, type2, N, fpcr);
        return result;

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

// FPMaxNum()
// =========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fp1cr)
  assert N IN {16,32,64};
  (type1,-,-) = FPUnpack(op1, fp1cr);
  (type2,-,-) = FPUnpack(op2, fp1cr);

  boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
  boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
  boolean altfp = HaveAltFP() && !UsingAArch32() && fp1cr.AH == '1';

  if !(altfp && type1_nan && type2_nan) then
    // Treat a single quiet-NaN as -Infinity.
      if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1');
      elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1');
  
      altfmaxfmin = FALSE;   // Restrict use of FMAX/FMIN NaN propagation rules
      result = FPMax(op1, op2, fp1cr, altfmaxfmin);
  
      return result;

library pseudocode for shared/functions/float/fpmerge/IsMerging

// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRType fp1cr)
  boolean merge = HaveAltFP() && !UsingAArch32() && fp1cr.NEP == '1';
  return merge;
Library pseudocode for shared/functions/float/fpmin/FPMin

// FPMin()
// ======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMin(op1, op2, fpcr, altfp);

// FPMin()
// ======

// Compare two operands and return the smaller operand after rounding. The 'fpcr'
// argument supplies the FPCR control bits and 'altfp' determines if the function
// should use alternative behaviour.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean altfp)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if (altfp && type1 == FPType_Zero && type2 == FPType_Zero &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);

    if !done then
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped
            // underflow exception for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';
            result = FPRound(value, fpcr, rounding, TRUE);
        FPProcessDenorms(type1, type2, N, fpcr);
    return result;
Library pseudocode for shared/functions/float/fpminnum/FPMInNum

```c
// FPMInNum()
// =========

bits(N) FPMInNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

if !(altfp && type1_nan && type2_nan) then
  // Treat a single quiet-NaN as +Infinity.
  if type1 == FPType_QNaN && type2 != FPType_QNaN then
    op1 = FPInfinity('0');
  elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
    op2 = FPInfinity('0');
  altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
  result = FPMIn(op1, op2, fpcr, altfmaxfmin);
else
  return result;
```

Library pseudocode for shared/functions/float/fpmul/FPMul

```c
// 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);
  FPProcessDenorms(type1, type2, N, fpcr);
else
  return result;
```
Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd
bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
  boolean fpexc = TRUE; // Generate floating-point exceptions
  return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// =========
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)
  assert N IN {16,32,64};
  (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
  (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
  (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
  rounding = FPRoundingMode(fpcr);
  inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
  inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
  (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

if !((HaveAltFP() && !UsingAArch32() && fpcr.AH == '1')) then
  if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  if !done then
    infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

    // Determine sign and type product will have if it does not cause an
    // Invalid Operation.
    signP = sign1 EOR sign2;
    infP  = inf1 || inf2;
    zeroP = zero1 || zero2;

    // Non SNaN-generated Invalid Operation cases are multiplies of zero
    // by infinity and additions of opposite-signed infinities.
    invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);
    if invalidop then
      result = FPDefaultNaN();
    if fpexc then
      FPProcessException(FPExc_InvalidOp, fpcr);
  elseif (infA && signA == '0') || (infP && signP == '0') then
    result = FPInfinity('0');
  elseif (infA && signA == '1') || (infP && signP == '1') then
    result = FPInfinity('1');

  // Cases where the result is exactly zero and its sign is not determined by the
  // rounding mode are additions of same-signed zeros.
  elseif zeroA && zeroP && signA == signP then
    result = FPZero(sigA);

  // Otherwise calculate numerical result and round it.
  else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr, rounding, fpexc);
    if !invalidop && fpexc then
      FPProcessDenorms3(typeA, type1, type2, N, fpcr);
return result;

Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
assert N == 32;
rounding = FPRoundingMode(fpcr);
(typeA,signA,valueA) = FPUnpack(addend, fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
(done, result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);

if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
  if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  if done then
    infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
    // Determine sign and type product will have if it does not cause an
    // Invalid Operation.
    signP = sign1 EOR sign2;
    infP = inf1 || inf2;
    zeroP = zero1 || zero2;
    // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
    // additions of opposite-signed infinities.
    invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);
    if invalidop then
      result = FPDefaultNaN();
      FPProcessException(FPExc_InvalidOp, fpcr);
    // Other cases involving infinities produce an infinity of the same sign.
    elsif (infA && signA == '0') || (infP && signP == '0') then
      result = FPInfinity('0');
    elsif (infA && signA == '1') || (infP && signP == '1') then
      result = FPInfinity('1');
    // Cases where the result is exactly zero and its sign is not determined by the
    // rounding mode are additions of same-signed zeros.
    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);
    if !invalidop then
      FPProcessDenorm(typeA, N, fpcr);
  return result;
Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

// FPProcessNaNs3H()
// ============

(boolean, bits(N)) FPProcessNaNs3H(
    FPTYPE type1, FPTYPE type2, FPTYPE type3,
    bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
    FPCRType fpcr)

    assert N IN {32,64};

    bits(N) result;
    // When TRUE, use alternative NaN propagation rules.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan = type1 IN {FPTYPE_SNaN, FPTYPE_QNaN};
    boolean op2_nan = type2 IN {FPTYPE_SNaN, FPTYPE_QNaN};
    boolean op3_nan = type3 IN {FPTYPE_SNaN, FPTYPE_QNaN};
    boolean fpexc = TRUE;
    if altfp then
        if (type1 == FPTYPE_SNaN || type2 == FPTYPE_SNaN || type3 == FPTYPE_SNaN) then
            type_nan = FPTYPE_SNaN;
        else
            type_nan = FPTYPE_QNaN;
        if altfp && op1_nan && op2_nan && op3_nan then
            done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc));  // <n> register NaN selected
        elsif altfp && op2_nan && (op1_nan || op3_nan) then
            done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc));  // <n> register NaN selected
        elsif altfp && op3_nan && op1_nan then
            done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpcr, fpexc));  // <m> register NaN selected
        elsif type1 == FPTYPE_SNaN then
            done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
        elsif type2 == FPTYPE_SNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc));
        elsif type3 == FPTYPE_SNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc));
        elsif type1 == FPTYPE_QNaN then
            done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
        elsif type2 == FPTYPE_QNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc));
        elsif type3 == FPTYPE_QNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc));
        else
            done = FALSE; result = Zeros();  // 'Don't care' result
    return (done, result);
// 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 == FPTypeInfinity);
  inf2 = (type2 == FPTypeInfinity);
  zero1 = (type1 == FPTypeZero);
  zero2 = (type2 == FPTypeZero);
  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);
  
  FPProcessDenorms(type1, type2, N, fpcr);
return result;

// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)
assert N IN {16,32,64};
if !UsingAArch32() &HaveAltFP() then
  FPCRType fpcr = FPCR[];
  if fpcr.AH == '1' then
    (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
    if fptype IN {FPType_SNan, FPType_QNaN} then
      return op;  // When fpcr.AH=1, sign of NaN has no consequence
    else
      return NOT(op<N-1>) : op<N-2:0>;
  
Library pseudocode for shared/functions/float/fpmulx/FPMulX

Library pseudocode for shared/functions/float/fpneg/FPNeg

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive
// FPProcessDenorm()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPType ftype, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPProcessException(FPExc_InputDenorm, fpcr);

// FPProcessDenorms()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms(FPType type1, type2, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);

// FPProcessDenorms3()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms3(FPType type1, type2, type3, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
                           type3 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);

// 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

// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// ==============
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr, boolean fpexc)
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;
    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```plaintext
// FPProcessNaNs()
// ===============

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2, FPCRType fpcr)

boolean altfmaxfmin = FALSE;  // Do not use alfp mode for FMIN, FMAX and variants
boolean fpexc = TRUE;         // Generate floating-point exceptions
return FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);

// FPProcessNaNs()
// ===============

// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behaviour for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2, FPCRType fpcr, boolean altfmaxfmin, boolean fpexc)

assert N IN {16,32,64};
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;
FPType type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

if altfmaxfmin && (op1_nan || op2_nan) then
    FPProcessException(FPExc_InvalidOp, fpcr);
done = TRUE; sign2 = op2<N-1>;
    result = if type2 == FPType_Zero then FPZero(sign2) else op2;
elsif altfp && op1_nan && op2_nan then
    done = TRUE; result = FPProcessNaN(type_nan, op1, fpcr, fpexc);  // <n> register NaN selected
elsif type1 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
elsif type1 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
else
    done = FALSE; result = Zeros();  // 'Don't care' result
return (done, result);
```

// FPProcessNaNs3()
// ===============
(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                          bits(N) op1, bits(N) op2, bits(N) op3,
                          FPCRType fpcr)
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPProcessNaNs3()
// ===============
The boolean part of the return value says whether a NaN has been found and
processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.

// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
deeply 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, boolean fpexc)
    assert N IN {16,32,64};
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp then
      if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
        type_nan = FPType_SNaN;
      else
        type_nan = FPType_QNaN;
    elsif altfp && op1_nan && op2_nan && op3_nan then
      done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);   // <n> register NaN selected
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
      done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);   // <n> register NaN selected
    elsif altfp && op3_nan && op1_nan then
      done = TRUE;  result = FPProcessNaN(type_nan, op3, fpcr, fpexc);   // <m> register NaN selected
    elsif type1 == FPType_SNaN then
      done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
      done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_SNaN then
      done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
      done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
      done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_QNaN then
      done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    else
      done = FALSE;  result = Zeros();   // 'Don't care' result
    return (done, result);
FPRecipEstimate() 
--------------------

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr) 
assert N IN {16,32,64};

// When using alternative floating-point behaviour, do not generate 
// floating-point exceptions, flush denormal input and output to zero, 
// and use RNE rounding mode.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

boolean fpexc = !altfp;
if altfp then fpcr.<FIZ,FZ> = '11';
if altfp then fpcr.RMode = '00';

(fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

FPRounding rounding = FPRoundingMode(fpcr);
if fptype == FPType_SNaN || fptype == FPType_QNaN then 
    result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elsif fptype == FPType_Infinity then 
    result = FPZero(sign);
elsif fptype == FPType_Zero then 
    result = FPInfinity(sign);
if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
elsif ((N == 16 && Abs(value) < 2.0^-16) ||
     (N == 32 && Abs(value) < 2.0^-128) ||
     (N == 64 && Abs(value) < 2.0^-1024)) then 
    case rounding of
    when FPRounding_TIEEVEN
        overflow_to_inf = TRUE;
    when FPRounding_POSINF
        overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
        overflow_to_inf = (sign == '1');
    when FPRounding_ZERO
        overflow_to_inf = FALSE;
    result = if overflow_to_inf then FPInfinity(sign) else FMaxNormal(sign);
if fpexc then
    FPProcessException(FPExc_Overflow, fpcr);
    FPProcessException(FPExc_Inexact, fpcr);
elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
    && (N == 16 && Abs(value) >= 2.0^14) ||
     (N == 32 && Abs(value) >= 2.0^126) ||
     (N == 64 && Abs(value) >= 2.0^1022)) then 
    // Result flushed to zero of correct sign 
    result = FPZero(sign);

    // Flush-to-zero never generates a trapped exception. 
if UsingAArch32() then 
    FPSR.UFC = '1';
else 
    if fpexc then FPSR.UFC = '1';
else 
    // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and 
    // calculate result exponent. Scaled value has copied sign bit, 
    // exponent = 1022 = double-precision biased version of -1, 
    // fraction = original fraction 
    case N of 
    when 16
        fraction = operand<9:0> : Zeros(42);
        exp = UInt(operand<14:10>);
    when 32
        fraction = operand<22:0> : Zeros(29);
        exp = UInt(operand<30:23>);
    when 64
        fraction = operand<51:0>;
    Shared Pseudocode Functions
exp = \text{UInt}(\text{operand}<62:52>);

if exp == 0 then
   if fraction<51> == '0' then
      exp = -1;
      fraction = fraction<49:0>:'00';
   else
      fraction = fraction<50:0>:'0';

integer scaled;
boolean increasedprecision = N==32 && \text{HaveFeatRPRES}() && altfp;

if !increasedprecision then
   scaled = \text{UInt}('1':fraction<51:44>);
else
   scaled = \text{UInt}('1':fraction<51:41>);

case N of
   when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 29+1 = 30
   when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254
   when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

// Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
// fixed-point number in range [0.5 .. 1.0].
estimate = \text{RecipEstimate}(scaled, increasedprecision);

// Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.
if !increasedprecision then
   fraction = estimate<7:0> : \text{Zeros}(44);
else
   fraction = estimate<11:0> : \text{Zeros}(40);

if result_exp == 0 then
   fraction = '1' : fraction<51:1>;
elsif result_exp == -1 then
   fraction = '01' : fraction<51:2>;
   result_exp = 0;

case N of
   when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
   when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
   when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;
return result;
Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

// RecipEstimate()
// ===============
// Compute estimate of reciprocal of 9-bit fixed-point number.
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a, boolean increasedprecision)

  integer r;
  if !increasedprecision then
    assert 256 <= a && a < 512; // Round to nearest
    a = a*2+1;
    integer b = (2^19) DIV a;
    r = (b+1) DIV 2; // Round to nearest
    assert 256 <= r && r < 512;
  else
    assert 2048 <= a && a < 4096; // Round to nearest
    a = a*2+1;
    real real_val = Real(2^25)/Real(a);
    r = RoundDown(real_val);
    real error = real_val - Real(r);
    boolean round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
    if round_up then r = r+1;
    assert 4096 <= r && r < 8192;
  return r;
Library pseudocode for shared/functions/float/fprecpx/FPRecpX

// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};

    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;
    end

    bits(N)           result;
    bits(esize)       exp;
    bits(esize)       max_exp;
    bits(N-(esize+1)) frac = Zeros();

    boolean altfp = HaveAltFP() & fpcr.AH == '1';
    boolean fpexc = !altfp;                 // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';     // Flush denormal input and output to zero
        (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
    end

    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>;
    end

    max_exp = Ones(esize) - 1;

    if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;
        end
    end

    return result;
// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    boolean isbfloat16  = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding, boolean fpexc)
    fpcr.AHP = '0';
    boolean isbfloat16  = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
// FPRoundBase()
// =============

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat16)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRoundBase()
// =============

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat16, boolean fpexc)

assert N IN {16,32,64};
assert op != 0.0;
assert rounding != FPRounding_TIEAWAY;
bits(N) result;

// Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
if N == 16 then
    minimum_exp = -14;  E = 5;  F = 10;
elsif N == 32 && isbfloat16 then
    minimum_exp = -126;  E = 8;  F = 7;
elsif N == 32 then
    minimum_exp = -126;  E = 8;  F = 23;
else  // N == 64
    minimum_exp = -1022;  E = 11;  F = 52;

// Split value into sign, unrounded mantissa and exponent.
if op < 0.0 then
    sign = '1';  mantissa = -op;
else
    sign = '0';  mantissa = op;
exponent = 0;
while mantissa < 1.0 do
    mantissa = mantissa * 2.0;  exponent = exponent - 1;
while mantissa >= 2.0 do
    mantissa = mantissa / 2.0;  exponent = exponent + 1;

// When TRUE, detection of underflow occurs after rounding and the test for a
// denormalized number for single and double precision values occurs after rounding.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

// Deal with flush-to-zero before rounding if FPCR.AH != '1'.
if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
    exponent < minimum_exp) then
    // Flush-to-zero never generates a trapped exception.
    if UsingAAArch32() then
        FPSCR.UFC = '1';
    else
        FPSR.UFC = '1';
    return FPZero(sign);

biased_exp_unconstrained = exponent - minimum_exp + 1;
int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
based_exp = Max(exponent - minimum_exp + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);
Underflow occurs if exponent is too small before rounding, and result is inexact or the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.

if |altfp & biased_exp == 0 & (error != 0.0 || fpcr.UFE == '1') then
  if fpexc then FProcessException(FPEX_Underflow, fpcr);

// Round result according to rounding mode.
if altfp then
  case rounding of
    when FPRounding_TIEEVEN
      round_up_unconstrained = (error_unconstrained > 0.5 ||
        (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
      round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
      overflow_to_inf = TRUE;
    when FPRounding_POSINF
      round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
      round_up = (error != 0.0 && sign == '0');
      overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
      round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
      round_up = (error != 0.0 && sign == '1');
      overflow_to_inf = (sign == '1');
    when FPRounding_ZERO, FPRounding_ODD
      round_up_unconstrained = FALSE;
      round_up = FALSE;
      overflow_to_inf = FALSE;

  if round_up_unconstrained then
    int_mant_unconstrained = int_mant_unconstrained + 1;
    if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
      biased_exp_unconstrained = biased_exp_unconstrained + 1;
      int_mant_unconstrained = int_mant_unconstrained DIV 2;

  // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
  if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
    // the result of unconstrained rounding is less than the minimum normalized number
    if fpcr.FZ == '1' & N != 16 || fpcr.FZ16 == '1' & N == 16 then // Flush-to-zero
      if fpexc then
        FPSR.UFC = '1';
        FProcessException(FPExc_Inexact, fpcr);
      end
      return FPZero(sign);
    elsif error != 0.0 || fpcr.UFE == '1' then
      if fpexc then FProcessException(FPEX_Underflow, fpcr);
    else // altfp == FALSE
      case rounding of
        when FPRounding_TIEEVEN
          round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
          overflow_to_inf = TRUE;
        when FPRounding_POSINF
          round_up = (error != 0.0 && sign == '0');
          overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
          round_up = (error != 0.0 && sign == '1');
          overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
          round_up = FALSE;
          overflow_to_inf = FALSE;
      end
      if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then // Rounded up from denormalized to normalized
          biased_exp = 1;
        if int_mant == 2^(F+1) then // Rounded up to next exponent
          biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;
      // Handle rounding to odd
      if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';
// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
      result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
      error = 1.0;  // Ensure that an Inexact exception occurs
    else                      // Alternative half precision
      result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
    else
      if biased_exp >= 2^E then
        result = sign : Ones(N-1);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        error = 0.0;  // Ensure that an Inexact exception does not occur
      else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

// Deal with Inexact exception.
if error != 0.0 then
    if fpexc then FPProcessException(FPExc_Inexact, fpcr);
return result;

Library pseudocode for shared/functions/float/fpround/FPRoundCV

// FPRoundCV()
// ===========
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.
bits(N) FPRoundCV(real op, FPCRType fpcr, FPRounding rounding)
  fpcr.FZ16 = '0';
  boolean fpexc = TRUE;  // Generate floating-point exceptions
  return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

Library pseudocode for shared/functions/float/fprounding/FPRounding

enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF, FPRounding_NEGINF, FPRounding_ZERO, FPRounding_TIEAWAY, FPRounding_ODD};

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.
FPRounding FPRoundingMode(FPCRType fpcr)
  return FPDecodeRounding(fpcr.RMode);
// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
assert rounding != FPRounding_ODD;
assert N IN {16,32,64};

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpxc = !altfp;

// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
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.
    result = Real(int_result);
    // Re-encode as a floating-point value, result is always exact.
    if real_result == 0.0 then
        result = FPZero(sign);
    else
        result = FPRound(real_result, fpcr, FPRounding_ZERO);
    // Generate inexact exceptions.
    if error != 0.0 && exact then
        FPProcessException(FPExc_Inexact, fpcr);
return result;
Library pseudocode for shared/functions/float/fproundintn/FPRoundIntN
bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
assert rounding != FPRounding_ODD;
assert N IN {32,64};
assert intsize IN {32, 64};
integer exp;
constant integer E = (if N == 32 then 8 else 11);
constant integer F = N - (E + 1);

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;

// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
    if N == 32 then
        exp = 126 + intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
    else
        exp = 1022+intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
            when FPRounding_POSINF
                round_up = error != 0.0;
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_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;
        overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);
        if overflow then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
                FPProcessException(FPExc_InvalidOp, fpcr);
            // This case shouldn't set Inexact.
            error = 0.0;
        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero(sign);
            else
                result = FPRound(real_result, fpcr, FPRounding_ZERO);
// 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};

  // When using alternative floating-point behaviour, do not generate
  // floating-point exceptions and flush denormal input to zero.
  boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
  boolean fpexc = !altfp;
  if altfp then fpcr.<FIZ,FZ> = '11';

  (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

  if fptype == FPType_SNaN || fptype == FPType_QNaN then
    result = FPProcessNaN(fptype, operand, fpcr, fpexc);
  elsif fptype == FPType_Zero then
    result = FPInfinity(sign);
    if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
  elsif sign == '1' then
    result = FPDefaultNaN();
    if fpexc then 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>);
    endcase

    if exp == 0 then
      while fraction<51> == '0' do
        fraction = fraction<50:0> : '0';
        exp = exp - 1;
      endwhile
      fraction = fraction<50:0> : '0';

      integer scaled;
      boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;
      if !increasedprecision then
        if exp<0> == '0' then
          scaled = UInt('1':fraction<51:44>);
        else
          scaled = UInt('01':fraction<51:45>);
        endelse
      else
        if exp<0> == '0' then
          scaled = UInt('1':fraction<51:41>);
        else
          scaled = UInt('01':fraction<51:42>);
        endelse
    endcase

    // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
    // fixed-point result in the range [1.0 .. 2.0].

    estimate = RecipSqrtEstimate(scaled, increasedprecision);
/ Convert to scaled floating point result with copied sign bit and high-order
/ fraction bits, and exponent calculated above.

\[
\text{case N of}
\begin{align*}
\text{when 16} & \quad \text{result} = '0' : \text{result\_exp}\langle N-12:0 \rangle : \text{estimate}\langle 7:0 \rangle : \text{Zeros}(2); \\
\text{when 32} & \quad \text{if !increasedprecision then} \\
& \quad \text{result} = '0' : \text{result\_exp}\langle N-25:0 \rangle : \text{estimate}\langle 7:0 \rangle : \text{Zeros}(15); \\
& \quad \text{else} \\
& \quad \text{result} = '0' : \text{result\_exp}\langle N-25:0 \rangle : \text{estimate}\langle 11:0 \rangle : \text{Zeros}(11); \\
\text{when 64} & \quad \text{result} = '0' : \text{result\_exp}\langle N-54:0 \rangle : \text{estimate}\langle 7:0 \rangle : \text{Zeros}(44); \\
\end{align*}
\]

\text{return result;}

// Shared Pseudocode Functions
Page 2792
// RecipSqrtEstimate()
// ===================
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
// a is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a, boolean increasedprecision)
integer r;
if !increasedprecision then
  assert 128 <= a && a < 512;
  if a < 256 then                      // 0.25 .. 0.5
    a = a*2+1;                       // a in units of 1/512 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
      b = b+1;
    // b = largest b such that b < 2^14 / sqrt(a)
    r = (b+1) DIV 2;                     // Round to nearest
  assert 256 <= r && r < 512;
  else
    assert 1024 <= a && a < 4096;
    real real_val;
    real error;
    integer int_val;
    if a < 2048 then                     // 0.25... 0.5
      a = a*2 + 1;                     // Take 10 bits of fraction and force a 1 at the bottom
      real_val = Real(a)/2.0;
    else                                 // 0.5..1.0
      a = (a >> 1) << 1;               // Discard bottom bit
      a = a+1;                         // Taking 10 bits of the fraction and force a 1 at the bottom
      real_val = Real(a);
    real_val = Sqrt(real_val);           // This number will lie in the range of 32 to 64
    real_val = real_val * Real(2^47);    // Round to nearest even for a DP float number
    int_val = RoundDown(real_val);      // The integer is the size of the whole DP mantissa
    error = real_val - Real(int_val);   // Calculate rounding value
    round_up = error > 0.5;              // Error cannot be exactly 0.5 so do not need tie case
    if round_up then int_val = int_val+1;
    real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
    int_val = RoundDown(real_val);      // Round that (to nearest even) to give integer
    error = real_val - Real(int_val);
    round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
    if round_up then int_val = int_val+1;
    r = int_val;
  assert 4096 <= r && r < 8192;
return r;
Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
assert N IN {16,32,64};
(fptype,sign,value) = FPUnpack(op, fpcr);

if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
  result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPTYPE_Zero then
  result = FPZero(sign);
elsif fptype == FPTYPE_Infinity && sign == '0' then
  result = FPIInfinity(sign);
elsif sign == '1' then
  result = FPDefaultNaN();
  FPProcessException(FPExc_InvalidOp, fpcr);
else
  result = FPRound(Sqrt(value), fpcr);
  FPProcessDenorm(fptype, N, fpcr);
return result;

Library pseudocode for shared/functions/float/fpsub/FPSub

// FPSub()
// ========

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPTYPE_Infinity);
  inf2 = (type2 == FPTYPE_Infinity);
  zero1 = (type1 == FPTYPE_Zero);
  zero2 = (type2 == FPTYPE_Zero);

  if inf1 && inf2 && sign1 == sign2 then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
    result = FPIInfinity('0');
  elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
    result = FPIInfinity('1');
  elsif zero1 && zero2 && sign1 == NOT(sign2) then
    result = FPZero(sign1);
  else
    result_value = value1 - value2;
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr, rounding);
    FPProcessDenorms(type1, type2, N, fpcr);
  return result;
Library pseudocode for shared/functions/float/fpthree/FPThree

```c
// FPThree()
// =========

bits(N) FPThree(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '1':Zeros(E-1);
frac = '1':Zeros(F-1);
result = sign : exp : frac;
return result;
```

Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```c
// 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;

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;

// Unpack using fpcr to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

// If NaN, set cumulative flag or take exception.
if fptype == FPTYPE_SNAN || fptype == FPTYPE_QNAN then
FPProcessException(FPExc_InvalidOp, fpcr);

// Scale by fractional bits and produce integer rounded towards minus-infinity.
value = value * 2.0^fbits;
int_result = RoundDown(value);
error = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment.
case rounding of
when FPRounding_TIEEVEN
round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
when FPRounding_POSINF
round_up = (error != 0.0);
when FPRounding_NEGINF
round_up = FALSE;
when FPRounding_ZERO
round_up = (error != 0.0 && int_result < 0);
when FPRounding_TIEAWAY
round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
if round_up then int_result = int_result + 1;

// Generate saturated result and exceptions.
(result, overflow) = SatQ(int_result, M, unsigned);
if overflow then
FPProcessException(FPExc_InvalidOp, fpcr);
elself error != 0.0 then
FPProcessException(FPExc_Inexact, fpcr);

return result;
```
Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```plaintext
// FPToFixedJS()
// =============
// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS((bits(M) op, FPCRType fpcr, boolean Is64)
assert M == 64 && N == 32;
    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !((HaveAltFP() && !UsingAArch32()) && fpcr.AH == '1');
    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc_idenorm);
    Z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    int_result = RoundDown(value);
    error = value - Real(int_result);
    // Determine whether supplied rounding mode requires an increment.
    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));
    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        Z = '0';
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        Z = '0';
    elsif fptype == FPType_Infinity then result = 0;
    return (result<N-1:0>, Z);
```

Library pseudocode for shared/functions/float/fptwo/FPTwo

```plaintext
// FPTwo()
// =======

bits(N) FPTwo(bit sign)
assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;
    return result;
```
Library pseudocode for shared/functions/float/fptype/FPTypen

enumeration FPTypen {
  FPTypen_Zero,
  FPTypen_Denormal,
  FPTypen_Nonzero,
  FPTypen_Infinity,
  FPTypen_QNaN,
  FPTypen_SNaN};

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

// FPUnpack()
// =========
(FPTypen, bit, real) FPUnpack(bits(N) fpval, FPCTRLType fpcr)
  fpcr.AHP = '0';
  boolean fpexc = TRUE;   // Generate floating-point exceptions
  (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
  return (fp_type, sign, value);

// FPUnpack()
// =========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPTypen, bit, real) FPUnpack(bits(N) fpval, FPCTRLType fpcr, boolean fpexc)
  fpcr.AHP = '0';
  (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
  return (fp_type, sign, value);
// FPUnpackBase()
// ==============

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)
boolean fpexc = TRUE; // Generate floating-point exceptions
(fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
return (fp_type, sign, value);

// FPUnpackBase()
// ==============

// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)

// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr, boolean fpexc)
assert N IN {16,32,64};

boolean altfp = HaveAltFP() && !UsingAArch32();
boolean fiz = altfp && fpcr.FIZ == '1';
boolean fz = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
if N == 16 then
  sign = fpval<15>;
  exp16 = fpval<14:10>;
  frac16 = fpval<9:0>;
  if IsZero(exp16) then
    if IsZero(frac16) || fpcr.FZ16 == '1' then
      fptype = FPType_Zero; value = 0.0;
    else
      fptype = FPType_Denormal; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
    else
      if IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
        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);
    endif
  else
    if IsZero(exp16) then
      if IsZero(frac32) then
        // Produce zero if value is zero.
        fptype = FPType_Zero; value = 0.0;
      elsif fz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
        fptype = FPType_Zero; value = 0.0;
      else
        if fpcr.FIZ==1 then
          // Check whether to raise Input Denormal floating-point exception.
          // fpcr.FIZ==1 does not raise Input Denormal exception.
          if fz then
            // Denormalized input flushed to zero
            if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
            else
              fptype = FPType_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
            endif
          else
            fptype = if IsZero(frac32) then
              FPType_Denormal; value = 2.0^1000000;
            else
              fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
            endif
          endif
        endif
      endif
    else
      if IsZero(exp32) then
        if IsZero(frac32) then
          // Produce zero if value is zero.
          fptype = FPType_Zero; value = 0.0;
        else
          // Flush-to-zero if FIZ==1 or AH,FZ==01
          fptype = FPType_Zero; value = 0.0;
        endif
      else
        if IsOnes(exp32) then
          if IsZero(frac32) then
            // Infinity or NaN in IEEE format
            fptype = FPType_Infinity; value = 2.0^1000000;
          else
            fptype = if frac32<22> == '1' then
              FPType_QNaN else FPType_SNaN;
          endif
        else
          fptype = FPType_Nonzero;
          value = 2.0^(UInt(exp32)-23) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
        endif
      endif
    endif
  endif
endif
else
  if IsZero(exp32) then
    if IsZero(frac32) then
      // Produce zero if value is zero.
      fptype = FPType_Zero; value = 0.0;
    else
      fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
    endif
  else
    if IsOnes(exp32) then
      if IsZero(frac32) then
        // Infinity or NaN in IEEE format
        fptype = FPType_Infinity; value = 2.0^1000000;
      else
        fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
      endif
    else
      fptype = FPType_Nonzero;
      value = 2.0^(UInt(exp32)-23) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
    endif
  endif
endif
endif
 Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

// FPUnpackCV()
// ============
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr)
  fpcr.FZ16 = '0';
  boolean fpexc = TRUE; // Generate floating-point exceptions
  (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
  return (fp_type, sign, value);

 Library pseudocode for shared/functions/float/fpzero/FPZero

// FPZero()
// =========

bits(N) FPZero(bit sign)
  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = N - (E + 1);
  exp = Zeros(E);
  frac = Zeros(F);
  result = sign : exp : frac;
  return result;
Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;
    return result;

Library pseudocode for shared/functions/integer/AddWithCarry

// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

<bits(N), bits(4)> AddWithCarry<bits(N) x, bits(N) y, bit carry_in>
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

// AArch64.BranchAddr()
// ==============
// Return the virtual address with tag bits removed for storing to the program counter.

<bits(64) AArch64.BranchAddr<bits(64) vaddress>
    assert !UsingAAArch32();
    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

```pseudocode
enumeration AccType {
    AccType_NORMAL,        // Normal loads and stores
    AccType_STREAM,        // Streaming loads and stores
    AccType_ATOMIC,        // Atomic loads and stores
    AccType_ORDERED,       // Load-Acquire and Store-Release
    AccType_ATOMIC,        // Atomic loads and stores
    AccType_ATOMICRW,      // Atomic Load-Acquire and Store-Release with atomic access
    AccType_ATOMICLS64,    // Atomic 64-byte loads and stores
    AccType_ATOMICLINES64, // Atomic LINES64 loads and stores
    AccType_UNPRIV,        // Load and store unprivileged
    AccType_IFETCH,        // Instruction fetch
    AccType_TTW,           // Translation table walk
    AccType_NONFAULT,      // Non-faulting loads
    AccType_CNOTFIRST,     // Contiguous FF load, not first element
    AccType_NV2REGISTER,   // MRS/MSR instruction used at EL1 and which is converted to a memory access that uses the EL2 translation regime
    AccType_DC,            // Data cache maintenance
    AccType_DC_UNPRIV,     // Data cache maintenance instruction used at EL0
    AccType_IC,            // Instruction cache maintenance
    AccType_DCZVA,         // DC ZVA instructions
    AccType_AT};           // Address translation
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```pseudocode
type AccessDescriptor is (  
    AccType acctype,  
    MPAMinfo mpam,  
    boolean page_table_walk,  
    boolean secondstage,  
    boolean s2fswalk,  
    integer level  
);
```

Library pseudocode for shared/functions/memory/AddrTop

```pseudocode
// AddrTop()  
// =========  
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".  
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.  
integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)  
    assert HaveEL(el);  
    regime = S1TranslationRegime(el);  
    if ELUsingAArch32(regime) then  
        // AArch32 translation regime.  
        return 31;  
    else  
        if EffectiveTBI(address, IsInstr, el) == '1' then  
            return 55;  
        else  
            return 63;  
)
```

Library pseudocode for shared/functions/memory/AddressDescriptor

```pseudocode
type AddressDescriptor is (  
    FaultRecord fault,  
    MemoryAttributes mematts,  
    FullAddress paddress,  
    bits(64) vaddress  
);
```
**Library pseudocode for shared/functions/memory/Allocation**

```plaintext
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**

```plaintext
// BigEndian()
// ===========

boolean BigEndian(AccType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        return SCTLR_EL2.EE == '1';
    else
        if UsingAArch32() then
            bigend = (PSTATE.E != '0');
        elseif PSTATE.EL == EL0 then
            bigend = (SCTLR[].E0E != '0');
        else
            bigend = (SCTLR[].EE != '0');
        return bigend;
```

**Library pseudocode for shared/functions/memory/BigEndianReverse**

```plaintext
// 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**

```plaintext
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**

```plaintext
// CreateAccessDescriptor()
// ===============

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype);
    accdesc.page_table_walk = FALSE;
    return accdesc;
```
Library pseudocode for shared/functions/memory/CreateAccessDescriptorTTW

```
// CreateAccessDescriptorTTW()
// ===========================
AccessDescriptor CreateAccessDescriptorTTW(AccType acctype, boolean secondstage,
  boolean s2fs1walk, integer level)
{
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype);
    accdesc.page_table.walk = TRUE;
    accdesc.s2fs1walk = s2fs1walk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;
}
```

Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DescriptorUpdate

```
type DescriptorUpdate is (
  boolean AF,                  // AF needs to be set
  AddressDescriptor descaddr   // Descriptor to be updated
)
```

Library pseudocode for shared/functions/memory/DeviceType

```
enumeration DeviceType {
  DeviceType_GRE,
  DeviceType_nGRE,
  DeviceType_nGnRE,
  DeviceType_nGnRnE
}
```

Library pseudocode for shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
{
  assert HaveEL(el);
  regime = S1TranslationRegime(el);
  assert(!ELUsingAArch32(regime));

  case regime of
    when EL1
      tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
      if HavePACExt() then
        tbid = if address<55> == '1' then TCR_EL1.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;
    end

  return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');
}
// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));
    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;
    return tcma;

Library pseudocode for shared/functions/memory/Fault

enumeration Fault {Fault_None,
    Fault_AccessFlag,
    Fault_Alignment,
    Fault_Background,
    Fault_Domain,
    Fault_Permission,
    Fault_Translation,
    Fault_AddressSize,
    Fault_SyncExternal,
    Fault_SyncExternalOnWalk,
    Fault_SyncParity,
    Fault_SyncParityOnWalk,
    Fault_AsyncParity,
    Fault_AsyncExternal,
    Fault_Debug,
    Fault_TLBConflict,
    Fault_BranchTarget,
    Fault_HWUpdateAccessFlag,
    Fault_Lockdown,
    Fault_Exclusive,
    Fault_ICacheMaint};
Library pseudocode for shared/functions/memory/FaultRecord

type FaultRecord is (Fault statuscode, // Fault Status
    AccType acctype,      // Type of access that faulted
    FullAddress ipaddress, // Intermediate physical address
    boolean s2fs1walk,    // Is on a Stage 1 translation table walk
    boolean write,        // TRUE for a write, FALSE for a read
    integer level,        // For translation, access flag and permission faults
    bit extflag,          // IMPLEMENTATION DEFINED syndrome for external aborts
    bits(4) 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


Library pseudocode for shared/functions/memory/FullAddress

type FullAddress is (bits(52) address,
    bit NS                  // '0' = Secure, '1' = Non-secure
)

Library pseudocode for shared/functions/memory/Hint_Prefetch

// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the 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

```java
enumeration MemType {MemType_Normal, MemType_Device};
```

Library pseudocode for shared/functions/memory/MemoryAttributes

```java
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

```java
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

```java
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```java
SpeculativeStoreBypassBarrierToPA();
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```java
SpeculativeStoreBypassBarrierToVA();
```

Library pseudocode for shared/functions/memory/TLBRecord

```java
type TLBRecord is (Permissions perms, bit nG, // '0' = Global, '1' = not Global bits(4) domain, // AArch32 only bit GP, // Guarded Page boolean contiguous, // Contiguous bit from translation table integer level, // AArch32 Short-descriptor format: Indicates Section/Page integer blocksize, // Describes size of memory translated in KBytes DescriptorUpdate descupdate, // [Armv8.1] Context for h/w update of table descriptor bit CnP, // [Armv8.2] TLB entry can be shared between different PEs AddressDescriptor addrdesc)
```

Library pseudocode for shared/functions/memory/Tag

```java
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];

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>; 

Library pseudocode for shared/functions/mpam/DefaultPMG

constant PMGtype DefaultPMG = 0<7:0>;
// GenMPAMcurEL
// ============
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccType accType)
bits(2) mpamel;
boolean validEL;
boolean securempam;
boolean InD = accType IN {AccType_IFETCH, AccType_IC};
if HaveEMPAMExt() then
  boolean secure = IsSecure();
  securempam = MPAM3_EL3.FORCE_NS == '0' && secure;
  if MPAMisEnabled() && (!secure || MPAM3_EL3.SDEFLT == '0') then
    if UsingAArch32() then
      (validEL, mpamel) = ELFromM32(PSTATE.M);
    else
      mpamel = PSTATE.EL;
    if validEL then
      return genMPAM(UInt(mpamel), InD, securempam);
  else
    securempam = IsSecure();
    if HaveMPAMExt() && MPAMisEnabled() then
      if UsingAArch32() then
        (validEL, mpamel) = ELFromM32(PSTATE.M);
      else
        validEL = TRUE;
        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' &&
      EL2Enabled() &&
      (( el == 0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
        ( HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0' )) ||
      ( 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 &&
      EL2Enabled() &&
      MPAMHCR_EL2.GSTAPP_PLK == '1' && HCR_EL2.TGE == '0');
    integer eff_el = if gstplk then 1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel = genPMG(eff_el, InD, perr);
    returnInfo.mpam_ns = if secure then '0' else '1';
    returnInfo.partid  = partidel;
    returnInfo.pmg     = groupel;
    return returnInfo;

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.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(integer MPAMn, boolean InD)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();
    if InD then
        case MPAMn of
            when 3 partid = MPAM3_EL3.PARTID_I;
            when 2 partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            when 1 partid = MPAM1_EL1.PARTID_I;
            when 0 partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
        else
            case MPAMn of
                when 3 partid = MPAM3_EL3.PARTID_D;
                when 2 partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
                when 1 partid = MPAM1_EL1.PARTID_D;
                when 0 partid = MPAM0_EL1.PARTID_D;
                otherwise partid = PARTIDtype UNKNOWN;
        endcase;
        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 MOD 4) * 16;

return vpmw<vpme_lsb +: 16>;
Library pseudocode for shared/functions/registers/BranchTo

// BranchTo()
// =========
// Set program counter to a new address, with a branch type
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _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_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);
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
assert N == 64 || (N == 32 & UsingAArch32());
return _PC<N-1:0>;

// SPSR[] - non-assignment form
// ============================

bits(N) SPSR[]
assert N == 64;

bits(N) result;
if UsingAArch32() then
    case PSTATE.M of
        when M32_FIQ result = SPSR_fiq<N-1:0>;
        when M32_IRQ result = SPSR_irq<N-1:0>;
        when M32_Svc result = SPSR_svc<N-1:0>;
        when M32_Monitor result = SPSR_mon<N-1:0>;
        when M32_Abort result = SPSR_abt<N-1:0>;
        when M32_Hyp result = SPSR_hyp<N-1:0>;
        when M32.Undef result = SPSR_und<N-1:0>;
        otherwise unreachable();
    end case;
else
    case PSTATE.EL of
        when EL1 result = SPSR_EL1<N-1:0>;
        when EL2 result = SPSR_EL2<N-1:0>;
        when EL3 result = SPSR_EL3<N-1:0>;
        otherwise unreachable();
    end case;
end if;
return;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(N) value
if UsingAArch32() then
    case PSTATE.M of
        when M32_FIQ SPSR_fiq = ZeroExtend(value);
        when M32_IRQ SPSR_irq = ZeroExtend(value);
        when M32_Svc SPSR_svc = ZeroExtend(value);
        when M32_Monitor SPSR_mon = ZeroExtend(value);
        when M32_Abort SPSR_abt = ZeroExtend(value);
        when M32_Hyp SPSR_hyp = ZeroExtend(value);
        when M32.Undef SPSR_und = ZeroExtend(value);
        otherwise unreachable();
    end case;
else
    case PSTATE.EL of
        when EL1 SPSR_EL1 = ZeroExtend(value);
        when EL2 SPSR_EL2 = ZeroExtend(value);
        when EL3 SPSR_EL3 = ZeroExtend(value);
        otherwise unreachable();
    end case;
end if;
return;
**Library pseudocode for shared/functions/system/ArchVersion**

```c
enumeration ArchVersion {
    ARMv8p0,
    ARMv8p1,
    ARMv8p2,
    ARMv8p3,
    ARMv8p4,
    ARMv8p5,
    ARMv8p6,
    ARMv8p7
};
```

**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.ExecutingBTIInstr();

// PSTATE.BTYPE defaults to 00 for instructions that do not explictly 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();
```
// 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;
        valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
    when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
        // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
        // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
        // AArch64, then these modes are EL1 modes.
        el = (if HaveEL(EL3) && HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1);
    when M32_User
        el = EL0;
    otherwise
        valid = FALSE;  // Passed an illegal mode value

if !valid then el = bits(2) UNKNOWN;
return (valid, el);
Library pseudocode for shared/functions/system/ELFromSPSR

```c
// ELFromSPSR()
// ============
// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL' is the Exception level decoded from 'spsr'.

(boolean, bits(2)) ELFromSPSR(bits(N) spsr)
if spsr<4> == '0' then                      // AArch64 state
  el = spsr<3:2>;
elseif HighestELUsingAArch32() then       // No AArch64 support
  valid = FALSE;
elseif !HaveEL(el) then                  // Exception level not implemented
  valid = FALSE;
elseif spsr<1> == '1' then               // M[1] must be 0
  valid = FALSE;
elseif el == EL0 && spsr<0> == '1' then  // for EL0, M[0] must be 0
  valid = FALSE;
elseif 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;
elseif 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

```c
// ELIsInHost()
// ============

boolean ELIsInHost(bits(2) el)
if !HaveVirtHostExt() || ELUsingAArch32(EL2) then
  return FALSE;
  case el of
    when EL3
      return FALSE;
    when EL2
      return HCR_EL2.E2H == '1';
    when EL1
      return FALSE;
    when EL0
      return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
    otherwise
      Unreachable();
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32

```c
// ELStateUsingAArch32()
// ================

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
  // See ELStateUsingAArch32() 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 using AArch64
elsif el == HighestEL() then
    return (TRUE, FALSE);                      // This is highest Exception level, so is using AArch64
else
    boolean aarch32 = boolean UNKNOWN;
    boolean known = TRUE;
    aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() || 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
        end if
    else
        aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1, EL0});
    end if
    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);

Library pseudocode for shared/functions/system/ELUsingAArch32

// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
return ELStateUsingAArch32K(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/ExceptionalOccurrenceTargetState

enumeration ExceptionalOccurrenceTargetState {
  AArch32_NonDebugState,
  AArch64_NonDebugState,
  DebugState
};

Library pseudocode for shared/functions/system/FIQPending

// Returns TRUE if there is any pending physical FIQ
boolean FIQPending();

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

// GetPSRFromPSTATE()
// ===============
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState)
  if UsingAArch32() && (targetELState IN {AArch32_NonDebugState, DebugState}) then
    assert N == 32;
  else
    assert N == 64;
  bits(N) spsr = Zeros();
  spsr<31:28> = PSTATE.<N,Z,C,V>;
  if HavePANExt() then spsr<22> = PSTATE.PAN;
  spsr<20> = PSTATE.IL;
  if PSTATE.nRW == '1' then // AArch32 state
    spsr<27> = PSTATE.Q;
    spsr<26:25> = PSTATE.IT<1:0>;
    if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
    if HaveDITExt() then
      if targetELState == AArch32_NonDebugState then
        spsr<21> = PSTATE.DIT;
      else //AArch64_NonDebugState or DebugState
        spsr<24> = PSTATE.DIT;
    end
    if targetELState IN {AArch64_NonDebugState, DebugState} then
      spsr<21> = PSTATE.DIT;
    end
    if targetELState IN {AArch64_NonDebugState, DebugState} then
      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;
  end
  return spsr;
Library pseudocode for shared/functions/system/HasArchVersion

// HasArchVersion()
// ================
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/system/HaveAArch32EL

// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;  // The Exception level is not implemented
    elsif !HaveAnyAArch32() then
        return FALSE;  // No Exception level can use AArch32
    elsif HighestELUsingAArch32() then
        return TRUE;  // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;  // The highest Exception level is using AArch64
    elsif el == EL0 then
        return TRUE;  // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/system/HaveAnyAArch32

// HaveAnyAArch32()
// ================
// 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_WFE

// Hint_WFE()
// =========
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.
Hint_WFE(integer localtimeout)
    if !IsEventRegisterSet() then
        ClearEventRegister();
    else
        trap = FALSE;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            if HaveTWEDExt() then
                sctlr = SCTLR[];
                trap = sctlr.nTWE == '0';
                target_el = EL1;
            else
                AArch64.CheckForWFxTrap(EL1, TRUE);
            if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
                // Check for traps described by the Hypervisor.
                if HaveTWEDExt() then
                    trap = HCR_EL2.TWE == '1';
                    target_el = EL2;
                else
                    AArch64.CheckForWFxTrap(EL2, TRUE);
            if !trap && HaveEl(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                if HaveTWEDExt() then
                    trap = SCR_EL3.TWE == '1';
                    target_el = EL3;
                else
                    AArch64.CheckForWFxTrap(EL3, TRUE);
            if trap && PSTATE.EL != EL3 then
                (delay_enabled, delay) = WFETrapDelay(target_el);    // (If trap delay is enabled, Delay amount)
                if !WaitForEventUntilDelay(delay_enabled, delay) then
                    // Event did not arrive before delay expired
                    AArch64.WFxTrap(target_el, TRUE);                // Trap WFE
                else
                    WaitForEvent(localtimeout);
        else
            AArch64.CheckForFpTrap(EL0, FALSE);
    else
        WaitForEvent(localtimeout);

Library pseudocode for shared/functions/system/Hint_WFI

// Hint_WFI()
// =========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.
Hint_WFI(integer localtimeout)
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch64.CheckForWFxTrap(EL1, FALSE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFxTrap(EL2, FALSE);
        if HaveEl(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL3, FALSE);
        WaitForInterrupt(localtimeout);
Library pseudocode for shared/functions/system/Hint_Yield

// Provides a hint that the task performed by a thread is of low importance so that it could yield to improve overall performance.
Hint_Yield();

Library pseudocode for shared/functions/system/IRQPending

// Returns TRUE if there is any pending physical IRQ
boolean IRQPending();

Library pseudocode for shared/functions/system/illegalExceptionReturn

// IllegalExceptionReturn()
// ========================
boolean IllegalExceptionReturn(bits(N) spsr)

// Check for illegal return:
// * To an unimplemented Exception level.
// * To EL2 in Secure state, when SecureEL2 is not enabled.
// * To EL0 using AArch64 state, with SPSR.M[0]==1.
// * To AArch64 state with SPSR.M[1]==1.
// * To AArch32 state with an illegal value of SPSR.M.
(valid, target) = ELFromSPSR(spsr);
if !valid then return TRUE;

// Check for return to higher Exception level
if UInt(target) > UInt(PSTATE.EL) then return TRUE;

spsr_mode_is_aarch32 = (spsr<4> == '1');

// Check for illegal return:
// * To EL1, EL2 or EL3 with register width specified in the SPSR different from the Execution state used in the Exception level being returned to, as determined by the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
// * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
// * To AArch64 state from AArch32 state (should be caught by above)
(known, target_el_is_aarch32) = ELUsingAArch32K(target);
assert known || (target == EL0 && !ELUsingAArch32(EL1));
if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

// Check for illegal return from AArch32 to AArch64
if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

// Check for illegal return to EL1 when HCR.TGE is set and when either of
// * SecureEL2 is enabled.
// * SecureEL2 is not enabled and EL1 is in Non-secure state.
if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
  if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
return FALSE;

Library pseudocode for shared/functions/system/InstrSet

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

InstructionSynchronizationBarrier();
Library pseudocode for shared/functions/system/InterruptPending

// InterruptPending()
// =================
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
{
    bit vIRQstatus = (if VirtualIRQPending() then '1' else '0') OR HCR_EL2.VI;
    bit vFIQstatus = (if VirtualFIQPending() then '1' else '0') OR HCR_EL2.VF;
    bits(3) v_interrupts = HCR_EL2.VSE : vIRQstatus : vFIQstatus;
    pending_physical_interrupt = (IRQPending() || FIQPending() ||
        IsPhysicalSErrorPending());
    pending_virtual_interrupt = !IsInHost() && ((v_interrupts AND HCR_EL2.<AMO,IMO,FMO>) != '000');
    return pending_physical_interrupt || pending_virtual_interrupt;
}

Library pseudocode for shared/functions/system/IsEventRegisterSet

// IsEventRegisterSet()
// ================
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise.

boolean IsEventRegisterSet()
{
    return EventRegister == '1';
}

Library pseudocode for shared/functions/system/IsHighestEL

// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
{
    return HighestEL() == el;
}

Library pseudocode for shared/functions/system/IsInHost

// IsInHost()
// ===========

boolean IsInHost()
{
    return ELIsInHost(PSTATE.EL);
}

Library pseudocode for shared/functions/system/IsPhysicalSErrorPending

// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();
Library pseudocode for shared/functions/system/IsSErrorEdgeTriggered

```plaintext
// IsSErrorEdgeTriggered()
// =======================
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered(bits(24) syndrome)
if HaveRASExt() then
    if HaveDoubleFaultExt() then
        return TRUE;
    if UsingAArch32() && syndrome[11:10] != '00' then
        return TRUE;
    if !UsingAArch32() && syndrome[23] == '0' && syndrome[5:0] != '000000' then
        return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

Library pseudocode for shared/functions/system/IsSecure

```plaintext
// IsSecure()
// ============
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
    return TRUE;
elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
    return TRUE;
return IsSecureBelowEL3();
```

Library pseudocode for shared/functions/system/IsSecureBelowEL3

```plaintext
// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differences 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
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```
Library pseudocode for shared/functions/system/IsSecureEL2Enabled

// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
  if HaveEL(EL2) && HaveSecureEL2Ext() then
    if HaveEL(EL3) then
      if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
        return TRUE;
      else
        return FALSE;
    else
      return IsSecure();
  else
    return FALSE;

Library pseudocode for shared/functions/system/IsSynchronizablePhysicalSErrorPending

// Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsSynchronizablePhysicalSErrorPending();

Library pseudocode for shared/functions/system/IsVirtualSErrorPending

// Returns TRUE if a virtual SError interrupt is pending.
boolean IsVirtualSErrorPending();

Library pseudocode for shared/functions/system/LocalTimeoutEvent

// Returns TRUE if a local timeout event is generated when the value of
// CNTVCT_EL0 equals or exceeds the threshold value for the first time.
// If the threshold value is less than zero a local timeout event will
// not be generated.
boolean LocalTimeoutEvent(integer localtimeout);

Library pseudocode for shared/functions/system/Mode_Bits


constant bits(5) M32_User = '10000';
constant bits(5) M32_FIQ = '10001';
constant bits(5) M32IRQ = '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';

Library pseudocode for shared/functions/system/PLOfEL

// 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;

Library pseudocode for shared/functions/system/PSTATE

ProcState PSTATE;
Library pseudocode for shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

Library pseudocode for shared/functions/system/ProcState

```
type ProcState is (  
  bits (1) N,       // Negative condition flag  
  bits (1) Z,       // Zero condition flag  
  bits (1) C,       // Carry condition flag  
  bits (1) V,       // Overflow condition flag  
  bits (1) D,       // Debug mask bit  
  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) UA0,     // 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  
  bits (8) IT,       // If-then bits, RES0 in CPSR  
  [AArch32 only]  
  bits (1) J,       // J bit, RES0  
  [AArch32 only, RES0 in SPSR and CPSR]  
  bits (1) T,       // T32 bit, RES0 in CPSR  
  [AArch32 only]  
  bits (1) E,       // Endianness bit  
  [AArch32 only]  
  bits (5) M         // Mode field  
  [AArch32 only]  
)
```

Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()  
// ================  
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)  
  it = spsr<15:10,26:25>;  
  // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set  
  // to zero or copied from the SPSR.  
  if PSTATE.IL == '1' then  
    if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';  
    else return it;  
  // The IT bits are forced to zero when they are set to a reserved value.  
  if !IsZero(it<7:4>) && !IsZero(it<3:0>) then  
    return '00000000';  
  // The IT bits are forced to zero when returning to A32 state, or when returning to an EL  
  // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.  
  itd = if PSTATE.EL == EL2 then HSCLTR.ITD else SCTLR.ITD;  
  if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then  
    return '00000000';  
  else  
    return it;
```

Library pseudocode for shared/functions/system/SCRType

```
type SCRType;
```
Library pseudocode for shared/functions/system/SCR_GEN

```c
// SCR_GEN[]
// =========

SCRTypSCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if HighestELUsingAArch32() then
        r = ZeroExtend(SCR);
    else
        r = SCR_EL3;
    return r;
```

Library pseudocode for shared/functions/system/SendEvent

```c
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.
SendEvent();
```

Library pseudocode for shared/functions/system/SendEventLocal

```c
// 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;
```
Library pseudocode for shared/functions/system/SetPSTATEFromPSR

// SetPSTATEFromPSR()
// ===============
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr)
   boolean from_aarch64 = !UsingAArch32();
   assert N == (if from_aarch64 then 64 else 32);
   PSTATE.SS = DebugExceptionReturnSS(spsr);
   ShouldAdvanceSS = FALSE;
   if IllegalExceptionReturn(spsr) then
      PSTATE.IL = '1';
      if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
      if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
      if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
      if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
      if HaveMTEExt() then PSTATE.TCO = 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<8>;
         if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
         if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
         if HaveUAOExt() then PSTATE.UAO = spsr<23>;
         if HaveDITExt() then PSTATE.DIT = spsr<24>;
         if HaveMTEExt() then PSTATE.TCO = spsr<25>;
   // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
   // the T bit is set to zero or copied from 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.O = spsr<27>;
      PSTATE.IT = RestoredITBits(spsr);
      ShouldAdvanceIT = FALSE;
      if HaveDITExt() then PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>);
      PSTATE.GE = spsr<19:16>;
      PSTATE.E = spsr<9>;
      PSTATE.<A,I,F> = spsr<8:6>;
      // No PSTATE.D in AArch32 state
      PSTATE.T = spsr<5>;
      // PSTATE.J is RESO
   else // AArch64 state
      PSTATE.<D,A,I,F> = spsr<9:6>;
      // No 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/ShouldAdvanceSS

boolean ShouldAdvanceSS;

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();

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

// Take any pending unmasked physical SError interrupt
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.
TakeUnmaskedSErrorInterrupts();

Library pseudocode for shared/functions/system/ThisInstr

bits(32) ThisInstr();

Library pseudocode for shared/functions/system/ThisInstrLength

integer ThisInstrLength();

Library pseudocode for shared/functions/system/Unreachable

Unreachable()
  assert FALSE;

Library pseudocode for shared/functions/system/UsingAArch32

// 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;

Library pseudocode for shared/functions/system/VirtualFIQPending

// Returns TRUE if there is any pending virtual FIQ
boolean VirtualFIQPending();

Library pseudocode for shared/functions/system/VirtualIRQPending

// Returns TRUE if there is any pending virtual IRQ
boolean VirtualIRQPending();
Library pseudocode for shared/functions/system/WaitForEvent

```c
// WaitForEvent()
// ==============
// PE suspends its operation and enters a low-power state if the
// Event Register is clear and, for WFET, there is no Local
// Timeout event when the WFET is executed.

WaitForEvent(integer localtimeout)
    if (!IsEventRegisterSet() || LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/system/WaitForInterrupt

```c
// WaitForInterrupt()
// ==================
// PE suspends its operation to enter a low-power state until
// a WFI wake-up event occurs, the PE is reset, and, for WFIT,
// a Local Timeout Event is generated.

WaitForInterrupt(integer localtimeout)
    if localtimeout < 0 then
        EnterLowPowerState();
    else
        if !LocalTimeoutEvent(localtimeout) then
            EnterLowPowerState();
    return;
```
Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraint ConstrainUnpredictable(Unpredictable which)
  case which of
    when Unpredictable_VMVR
      return Constraint_UNDEF;
    when Unpredictable_WBOVERLAPLD
      return Constraint_WBSUPPRESS; // return loaded value
    when Unpredictable_WBOVERLAPST
      return Constraint_NONE; // store pre-writeback value
    when Unpredictable_LDPOVERLAP
      return Constraint_UNDEF; // instruction is UNDEFINED
    when Unpredictable_BASEOVERLAP
      return Constraint_NONE; // use original address
    when Unpredictable_DATAOVERLAP
      return Constraint_NONE; // store original value
    when Unpredictable_DEVPAGE2
      return Constraint_FAULT; // take an alignment fault
    when Unpredictable_DEVICETAGSTORE
      return Constraint_NONE; // Do not take a fault
    when Unpredictable_INSTRDEVICE
      return Constraint_NONE; // Do not take a fault
    when Unpredictable_RESCPACR
      return Constraint_TRUE; // Map to UNKNOWN value
    when Unpredictable_RESMAIR
      return Constraint_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable_RESTEXCR
      return Constraint_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable_RESTnSZ
      return Constraint_FORCE; // Map to the limit value
    when Unpredictable_OORTnSZ
      return Constraint_FORCE; // Map to the limit value
    when Unpredictable_LARGEIPA
      return ConstraintFORCE; // Restrict the input size to the PAMax value
    when Unpredictable_ESRCONDPASS
      return Constraint_TRUE; // 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_TRUE; // No match on Data Abort or Prefetch abort
    when Unpredictable_WPMASKANDRAS
      return Constraint_FALSE; // Watchpoint disabled
    when Unpredictable_WPBASCONTOUS
      return Constraint_FALSE; // Watchpoint disabled
    when Unpredictable_WSPWMASK
      return Constraint_DISABLED; // Watchpoint disabled
    when Unpredictable_WSPWMSKEDBITS
      return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_RESBWPCTRL
return Constraint.Disabled; // Breakpoint/watchpoint disabled
when Unpredictable_BPNOTIMPL
return Constraint.Disabled; // Breakpoint disabled
when Unpredictable_RESBPTYPE
return Constraint.Disabled; // Breakpoint disabled
when Unpredictable_BPNOTCTXCMP
return Constraint.Disabled; // Breakpoint disabled
when Unpredictable_BPMATCHHALF
return Constraint.False; // No match
when Unpredictable_BPISMATCHHALF
return Constraint.False; // Do not force alignment
when Unpredictable_RESTARTALIGNPC
return Constraint.True; // Force zero extension
when Unpredictable_ZEROUPPER
return Constraint.True; // zero top halves of X registers
when Unpredictable_EMUZEROUPPERPC
return Constraint.True; // zero top half of PC
when Unpredictable_A32FORCEALIGNPC
return Constraint.False; // Do not force alignment
when Unpredictable_SM
return Constraint.Undef; // disabled SMC is Unallocated
when Unpredictable_NONFAULT
return Constraint.False; // Speculation enabled
when Unpredictable_SVEZEROUPPER
return Constraint.True; // zero top bits of Z registers
when Unpredictable_SVELDNFDATA
return Constraint.True; // Load mem data in NF loads
when Unpredictable_SVELDNFZERO
return Constraint.True; // Write zeros in NF loads
when Unpredictable_CHECKSPNONEACTIVE
return Constraint.True; // Check SP alignment
when Unpredictable_AFUPDATE
return Constraint.False; // AF update for alignment or permission fault
when Unpredictable_IESBinDebug
return Constraint.True; // Use SCTLR[].IESB in Debug state
when Unpredictable_BADMESCRC
return Constraint.True; // Bad settings for PMSFCR_EL1/PMSEVFR_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
when Unpredictable_ALUEXCEPTIONRETURN
return Constraint.Undef; // Do not force alignment
when Unpredictable_DBGxVR_RESS
return Constraint.False; // WFxT in Debug state does not execute as a NOP
when Unpredictable_LS64UNSUPPORTED
return Constraint.LimitedAtomicity; // Accesses are not single-copy atomic above the byte level.
Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

// ConstrainUnpredictableBits()
// ============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONstrained.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);
    if c == ConstraintUNKNOWN then
        return (c, Zeros(width));           // See notes; this is an example implementation only
    else
        return (c, bits(width) UNKNOWN);    // bits result not used

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 == ConstraintUNKNOWN 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

```c
// General
Constraint_NONE, // Instruction executes with no change or side-effect to its described behavior
Constraint_UNKNOW, // 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,
Constraint_FALSE,
Constraint_DISABLED,
Constraint_UNCOND, // Instruction executes unconditionally
Constraint_COND, // Instruction executes conditionally
Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
// Load-store
Constraint_WBSUPPRESS,
Constraint_FAULT,
Constraint_LIMITED_ATOMICITY, // Accesses are not single-copy atomic above the byte level
// IPA too large
Constraint_FORCE, Constraint_FORCENOSLCHECK};
```
Library pseudocode for shared/functions/unpredictable/Unpredictable
Unpredictable { // VMSR on MVFR
  Unpredictable_VMSR,
  // Writeback/transfer register overlap (load)
  Unpredictable_WBOVERLAPLD,
  // Writeback/transfer register overlap (store)
  Unpredictable_WBOVERLAPST,
  // Load Pair transfer register overlap
  Unpredictable_LPDPOVERLAP,
  // Store-exclusive base/status register overlap
  Unpredictable_BASEOVERLAP,
  // Store-exclusive data/status register overlap
  Unpredictable_DATAOVERLAP,
  // Load-store alignment checks
  Unpredictable_DEVPAGE2,
  // Instruction fetch from Device memory
  Unpredictable_INSTRDEVICE,
  // Reserved CPACR value
  Unpredictable_RESCPACR,
  // Reserved MAIR value
  Unpredictable_RESMAIR,
  // Reserved TEX:C:B value
  Unpredictable_RESTEXCB,
  // Reserved PRRR value
  Unpredictable_RESPRRR,
  // Reserved DACR field
  Unpredictable_RESDACR,
  // Reserved VTCR.S value
  Unpredictable_RESVTCRS,
  // Reserved TCR.TnSZ value
  Unpredictable_RESTnSZ,
  // Reserved SCTLR_ELx.TCF value
  Unpredictable_RESTCF,
  // Tag stored to Device memory
  Unpredictable_DEVICETAGSTORE,
  // Out-of-range TCR.TnSZ value
  Unpredictable_OORTnSZ,
  // IPA size exceeds PA size
  Unpredictable_LARGEIPA,
  // Syndrome for a known-passing conditional A32 instruction
  Unpredictable_ESRCONDPASS,
  // Illegal State exception: zero PSTATE.IT
  Unpredictable_ILZEROIT,
  // Illegal State exception: zero PSTATE.T
  Unpredictable_ILZEROT,
  // Debug: prioritization of Vector Catch
  Unpredictable_BPVECTORCATCHPRI,
  // Debug Vector Catch: match on 2nd halfword
  Unpredictable_VCMATCHHALF,
  // Debug Vector Catch: match on Data Abort or Prefetch abort
  Unpredictable_VCMATCHDAPA,
  // Debug breakpoints and watchpoints: reserved control bits
  Unpredictable_BPNOTIMPL,
  // Debug breakpoints: reserved type
  Unpredictable_BPMISMATCHHALF,
  // Debug breakpoints: not-context-aware breakpoint
  Unpredictable_BPMISMATCHHALF,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONACTIVE,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Consider SCTLR[].IESB in Debug state
Unpredictable_IESBinDebug,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
// WFET or WFIT instruction in Debug state
Unpredictable_WFxTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Compare DBGxVR_RESS for BP/WP
Unpredictable_DBGxVR_RESS};
Library pseudocode for shared/functions/vector/AdvSIMDEncryptImm

// AdvSIMDEncryptImm()
// ================

bits(64) AdvSIMDEncryptImm(bit op, bits(4) cmode, bits(8) imm8)
  case cmode<3:1> of
    when '000'
      imm64 = Replicate(Zeros(24):imm8, 2);
    when '001'
      imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
    when '010'
      imm64 = Replicate(imm8:Zeros(24), 2);
    when '011'
      imm64 = Replicate(imm8:Zeros(8), 4);
    when '100'
      imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
    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);
      if cmode<0> == '0' & op == '1' then
        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);
      if cmode<0> == '1' & op == '0' then
        imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<5>:0>:Zeros(19);
        imm64 = Replicate(imm32, 2);
      if cmode<0> == '1' & op == '1' then
        if UsingAArch32() then ReservedEncoding();
        imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<5>:0>:Zeros(48);
  return imm64;

Library pseudocode for shared/functions/vector/MatMulAdd

// MatMulAdd()
// ===========

// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
  assert N == 128;
  bits(N) result;
  bits(32) sum;
  integer prod;
  for i = 0 to 1
    for j = 0 to 1
      sum = Int(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 == 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)
    estimate = RecipSqrtEstimate(UInt(operand<31:23>), FALSE);
    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);
  return result;
```
**Library pseudocode for shared/functions/vector/UnsignedRecipEstimate**

// UnsignedRecipEstimate()
// ================

bits(N) UnsignedRecipEstimate(bits(N) operand)
  assert N == 32;
  if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
    result = Ones(N);
  else
    // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)
    estimate = RecipEstimate(UInt(operand<31:23>), FALSE);
    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);
  return result;

**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<N-1:0>, 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' || InternalSecureNoninvasiveDebugEnabled());
  else
    // If no EL3, IsSecure() returns the Effective value of (SCR_EL3.NS == '0')
    niden = (!IsSecure() || InternalSecureNoninvasiveDebugEnabled());
  return (EDSCR.TFO == '0' || !niden);
Library pseudocode for shared/trace/selfhosted/TraceAllowed

// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the current Security state and Exception Level

boolean TraceAllowed()
    if !HaveTraceExt() then return FALSE;
    if SelfHostedTraceEnabled() then
        if IsSecure() && HaveEL(EL3) then
            secure_trace_enable = (if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE);
            if secure_trace_enable == '0' then return FALSE;
            TGE_bit = if EL2Enabled() then HCR_EL2.TGE else '0';
            case PSTATE.EL of
                when EL3 TRE_bit = if HighestELUsingAArch32() then TRFCR.E1TRE else '0';
                when EL2 TRE bit = TRFCR_EL2.E2TRE;
                when EL1 TRE_bit = TRFCR_EL1.E1TRE;
                when EL0 TRE bit = if TGE_bit == '1' then TRFCR_EL2.E0HTRE else TRFCR_EL1.E0TRE;
            return TRE_bit == '1';
        else
            return (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled());
    else
        return (!IsSecure() || !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();
// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
if SelfHostedTraceEnabled() then
    if HaveEL(EL2) then
        TS_el2 = TRFCR_EL2.TS;
        if TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP);
        case TS_el2 of
            when '00'  // Falls through to check TRFCR_EL1.TS
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt();
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable();  // ConstrainUnpredictableBits removes this case
        end_case
        TS_el1 = TRFCR_EL1.TS;
        if TS_el1 == 'x0' then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP);
        case TS_el1 of
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt();
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable();  // ConstrainUnpredictableBits removes this case
        end_case
else
    return TimeStamp_CoreSight;
end_if
Library pseudocode for shared/translation/attrs/CombineS1S2AttrHints

```c
// CombineS1S2AttrHints()
// ======================
// Combines cacheability attributes and allocation hints from stage 1 and stage 2

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc, AccType s2acctype)
{
    MemAttrHints result;
    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
    if apply_force_writeback then
        if S2CacheDisabled(s2acctype) then
            result.attrs = MemAttr_NC;  // force Non-cacheable
        elsif s2desc.attrs == '11' then
            result.attrs = s1desc.attrs;
        elsif s2desc.attrs == '10' then
            result.attrs = MemAttr_WB;  // force Write-back
        else
            result.attrs = MemAttr_NC;
        end
    else
        if s2desc.attrs == '01' || s1desc.attrs == '01' then
            result.attrs = bits(2) UNKNOWN;  // Reserved
        elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
            result.attrs = MemAttr_NC;  // Non-cacheable
        elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
            result.attrs = MemAttr_WT;  // Write-through
        else
            result.attrs = MemAttr_WB;  // Write-back
        end
    end
    if result.attrs == MemAttr_NC then
        result.hints = MemHint_No;
    elsif apply_force_writeback then
        if s1desc.attrs != MemAttr_NC then
            result.hints = s1desc.hints;
        else
            result.hints = MemHint_RWA;
        end
    else
        result.hints = s1desc.hints;
        result.transient = s1desc.transient;
    end
    return result;
}
```

Library pseudocode for shared/translation/attrs/CombineS1S2Device

```c
// 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/longconvertattrsints

// 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 accype)
    assert !IsZero(attrfield);
    MemAttrHints result;
    if S1CacheDisabled(accctype) then // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elsif attrfield<3:0> == '0100' then // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elsif attrfield<3:2> == '01' then // Write-back transient
            result.attrs = MemAttr_WB;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        else // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;
    return result;

library pseudocode for shared/translation/attrs/memattrdefaults

// MemAttrDefaults()
// =================
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes mematts)
    if mematts.memtype == MemType_Device then
        mematts.inner = MemAttrHints UNKNOWN;
        mematts.outer = MemAttrHints UNKNOWN;
        mematts.shareable = TRUE;
        mematts.outershareable = TRUE;
    else
        mematts.device = DeviceType UNKNOWN;
        if mematts.inner.attrs == MemAttr_NC & mematts.outer.attrs == MemAttr_NC then
            mematts.shareable = TRUE;
            mematts.outershareable = TRUE;
    return mematts;
Library pseudocode for shared/translation/attrs/S1CacheDisabled

// S1CacheDisabled()
// ===============

boolean S1CacheDisabled(AccType acctype)
if ELUsingAArch32(S1TranslationRegime()) then
    if PSTATE.EL == EL2 then
        enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
return enable == '0';

Library pseudocode for shared/translation/attrs/S2AttrDecode

// S2AttrDecode()
// ===============
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)
MemoryAttributes memattrs;
apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';

// Device memory
if (apply_force_writeback && attr<2> == '0') || attr<3:2> == '00' then
    memattrs.memtype = MemType_Device;
    case attr<1:0> of
        when '00'  memattrs.device = DeviceType_nGnRnE;
        when '01'  memattrs.device = DeviceType_nGnRE;
        when '10'  memattrs.device = DeviceType_nGRE;
        when '11'  memattrs.device = DeviceType_GRE;

// Normal memory
elsif apply_force_writeback then
    if attr<2> == '1' then
        memattrs.memtype = MemType_Normal;
        memattrs.inner.attrs = attr<1:0>;
        memattrs.outer.attrs = attr<1:0>;
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    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
    end
else
    memattrs = MemoryAttributes UNKNOWN;    // Reserved
return MemAttrDefaults(memattrs);

Library pseudocode for shared/translation/attrs/S2CacheDisabled

// S2CacheDisabled()
// ===============

boolean S2CacheDisabled(AccType acctype)
if ELUsingAArch32(EL2) then
    disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
else
    disable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
return disable == '1';
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 attr != '00';
MemAttrHints result;
if S2CacheDisabled(acctype) then // Force Non-cacheable
  result.attrs = MemAttr_NC;
  result.hints = MemHint_No;
else
  case attr of
    when '01'                          // Non-cacheable (no allocate)
      result.attrs = MemAttr_NC;
      result.hints = MemHint_No;
    when '10'                          // Write-through
      result.attrs = MemAttr_WT;
      result.hints = MemHint_RWA;
    when '11'                          // Write-back
      result.attrs = MemAttr_WB;
      result.hints = MemHint_RWA;
  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_RA;
    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

// WalkAttrDecode()
// ================

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)

MemoryAttributes memattrs;
AccType acctype = AccType_NORMAL;
memattrs.memtype = MemType_Normal;
memattrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
memattrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
memattrs.shareable = SH<1> == '1';
memattrs.outershareable = SH == '10';
memattrs.tagged = FALSE;
return MemAttrDefaults(memattrs);

Library pseudocode for shared/translation/translation/HasS2Translation

// HasS2Translation()
// ==================
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
return (EL2Enabled() && !IsInHost() && PSTATE.EL IN {EL0,EL1});

Library pseudocode for shared/translation/translation/Have16bitVMID

// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID";

Library pseudocode for shared/translation/translation/PAMax

// PAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()
return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
Library pseudocode for shared/translation/translation/S1TranslationRegime

// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;
    end

Library pseudocode for shared/translation/translation/VAMax

// VAMax()
// ========
// Returns the IMPLEMENTATION DEFINED upper limit on the virtual address
// size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T23:12
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.